В. А. Ильина, П. К. Силаев

ЧИСЛЕННЫЕ МЕТОДЫ ДЛЯ ФИЗИКОВ-ТЕОРЕТИКОВ

I

Москва ♦ Ижевск

2003

УДК 519.6

Ильина В. А., Силаев П. К.
Численные методы для физиков-теоретиков. I. — Москва-Ижевск: Институт компьютерных исследований, 2003, 132 стр.

Данное пособие основано на лекциях и практических занятиях по курсу численных методов для будущих физиков-теоретиков. Основная цель книги состоит в рассмотрении понятных и достаточно простых в написании алгоритмов, ориентированных главным образом на решение типичных задач теоретической физики и являющихся, безусловно, необходимой частью арсенала любого физика-теоретика.

Для студентов физических специальностей.

ISBN 5-93972-231-8

http://rcd.ru

Оглавление

1. Предисловие

Пособие основано на курсе лекций и практических занятий по курсу численных методов, читаемых студентам кафедры квантовой теории и физики высоких энергий физического факультета МГУ.

Отбор материала производился с учетом специализации кафедры — кафедра готовит физиков-теоретиков в области квантовой теории поля, физики частиц, теории гравитации и астрофизики, и на основе личного опыта работы авторов в этих областях.

Однако, отбирая материал для курса, авторы с удивлением обнаружили, что в арсенале физика-теоретика должен присутствовать практически весь «джентльментский набор» стандартных (простейших) алгоритмов. Специфика курса заключается разве что в перераспределении материала между разделами (например, очень малое внимание уделяется алгоритмам обработки и фильтрации данных; впрочем, этим вопросам и так посвящено огромное количество прекрасных руководств, к которым мы и отсылаем читателей). Кроме того, все изложение алгоритмов ориентировано именно на решение типичных задач теоретической физики.

Следует подчеркнуть, что целью авторов было дать пусть и не самые эффективные, но достаточно понятные и простые в написании алгоритмы. Опыт показывает, что коэффициент полезного действия алгоритма определяется не только его эффективностью, но и временем, необходимым для его реализации и отладки, а также временем, которое требуется для того, чтобы понять, почему в данной задаче он не срабатывает и как его следует модифицировать.

Авторы (наивно) предполагают знакомство с курсом математики, читаемом на физическом факультете МГУ. Поэтому никаких пояснений или доказательств, относящихся к собственным векторам, интегральным суммам или классическим ортогональным полиномам в пособии не приведено. Авторы старались создать максимально компактное «руководство к действию», снабженное лишь самыми необходимыми доказательствами или

пояснениями. Единственная избыточность в пособии связана с алгоритмами, которые сами авторы оценивают как неэффективные. Они приведены просто для того, чтобы при дальнейшем знакомстве с литературой по численным методам они не воспринимались как последнее слово науки и не возникал соблазн их использовать.

Также следует подчеркнуть, что пособие является почти целиком компилятивным. Вкладом авторов является главным образом отбор материала и оценки эффективности тех или иных алгоритмов (или вариантов алгоритмов), основанные на личном опыте. Оценки эти ни в коем случае не являются абсолютными — они относятся лишь к тем классам задач, с которыми сталкивались авторы. Эта ситуация достаточно типична: достаточно сравнить противоположные друг другу рекомендации о решении задачи Коши для дифференциальных уравнений в частных производных, данные в книгах Самарского и Тьюкольского и др., — эти рекомендации явно определялись теми типами задач, которые чаще попадались авторам.

Ссылки на литературу не претендуют ни на полноту, ни на приоритетную корректность. Эта ситуация опять-таки достаточно типична: как известно, автором алгоритма FFT являются отнюдь не Danielson and Lanczos, на которых принято ссылаться, а Гаусс.

2. Сортировка

Постановка задачи такова: имеет место одномерный массив, для элементов этого массива можно определить операцию сравнения «меньше/больше» (при этом совершенно неважна природа элементов массива и операции сравнения). Требуется упорядочить массив в порядке возрастания (или убывания). Существует огромное количество алгоритмов сортировки, большую часть которых применять не следует.

2.1. Прямое упорядочение

Алгоритм:

> Для всех i от 1 до $n-1$ проделываем следующее:
>
> Находим минимум среди $a_i \ldots a_n$ и меняем его местами с a_i.

Число операций пропорционально n^2. Метод очень легко реализовать, и если Вам нужно один раз отсортировать массив не очень большой длины (до $n \approx 1000$), то им вполне можно пользоваться.

2.2. Метод пузырька

Алгоритм:

Повторяем n раз следующее:

для всех i от 1 до $n-1$

упорядочиваем соседние элементы a_i и a_{i+1}.

Если при очередном прогоне по i от 1 до $n-1$ перестановок не было, то завершаем работу.

Число операций в общем случае пропорционально n^2. Однако на частично упорядоченном массиве (например: $5, 1, 2, 3, 4$) будет всего n операций, поскольку элемент 5 («пузырек») «всплывет» уже после первого прогона. Поскольку вероятность выигрыша по скорости очень мала, то лучше этот метод не применять.

2.3. Ракушечный метод I

Алгоритм:

Упорядочиваем пару элементов (a_1,a_n).

Упорядочиваем пары элементов (a_1,a_{n-1}) и (a_2,a_n).

Упорядочиваем пары элементов (a_1,a_{n-2}), (a_2,a_{n-1}) и (a_3,a_n).

...

Упорядочиваем пары элементов $(a_1, a_2), a_2, a_3), \ldots, (a_{n-1}, a_n)$.

Число операций пропорционально n^2. Этот метод лучше не применять.

2.4. Ракушечный[1] метод II

Алгоритм:

Упорядочиваем пары элементов $(a_1, a_{n/2})$, $(a_2, a_{n/2+1}) \ldots (a_{n/2-1}, a_n)$.

С учетом предыдущего упорядочивания упорядочиваем четверки элементов $(a_1, a_{n/4}, a_{n/2}, a_{3n/4})$; $(a_2, a_{n/4+1}, a_{n/2+1}, a_{3n/4+1})$; ...; $(a_{n/4-1}, a_{n/2-1}, a_{3n/4-1}, a_n)$.

Далее то же с восьмерками чисел.

...

[1]На самом деле автором этого метода был человек по фамилии Shell.

Наконец, упорядочиваем (a_1,a_2,a_3,...,a_n) (тоже с учетом предыдущего упорядочения).

Число операций пропорционально $n^{3/2}$. Этот метод лучше не применять.

2.5. Двоичная вставка

Алгоритм:

Упорядочиваем пару элементов (a_1,a_2).

Находим место для a_3 так, чтобы элементы a_1, a_2, a_3 были упорядочены.

...

Находим место для a_m так, чтобы элементы $a_1, a_2, \ldots, a_m$ были упорядочены. Место ищется с помощью «деления отрезка пополам»:

Поскольку $a_1, \ldots, a_{m-1}$ уже упорядочены, то сначала сравниваем элемент a_m с a_1 и a_{m-1}. Если он не лежит между ними, то переставляем его на первое или последнее место. Если он лежит между ними, то его место находится где-то на отрезке между индексами 1 и $m-1$. Делим этот отрезок пополам и выясняем, на какую из половинок он должен попасть. Для этого сравниваем a_m с $a_{m/2}$. Соответствующую половинку снова делим пополам, и т.д. Когда длина отрезка, на котором должен лежать a_m, сократится до 1, выяснится истинное место для a_m.

Переставляем a_m на это место.

...

Наконец, находим место для a_n так, чтобы элементы $a_1, a_2, \ldots a_n$ были упорядочены.

Заметим, что количество сравнений, нужное, чтобы найти правильное место для элемента a_m, есть $\log m$. Поэтому операций сравнения в этом алгоритме — $n \log n$. К сожалению, перестановка a_m на правильное место занимает m операций, так что полное число операций — n^2. Этот метод можно применять только в том случае, когда операция сравнения проходит очень медленно, т. е. если время, которое тратится на $n \log n$ сравнений, превышает время, которое тратится на n^2 перестановок.

2.6. Индексация

Иногда элементам массива можно приписать вес $\rho_i = \rho(a_i)$, который достаточно равномерно распределен по интервалу от минимума ρ_{min} до максимума ρ_{max} (например, мы сортируем n действительных чисел, лежащих на отрезке $(0, 10)$, причем количество чисел, попадающих на любой отрезок единичной длины, приблизительно равно $n/10$). В этом случае каждому элементу можно приписать целочисленный индекс $z_i = z(\rho_i) = (\rho_i - \rho_{min})/\Delta\rho$, где $\Delta\rho = (\rho_{max} - \rho_{min})/m$. Индекс z_i пробегает значения от 0 до $m - 1$, т. е. все элементы разбиваются на m групп. Количество элементов в группах приблизительно равно n/m. После этого все элементы индекса 0 переставляются в начало массива и упорядочиваются методом прямого упорядочения. Затем в начало оставшейся части массива переставляются все элементы индекса 1 и тоже упорядочиваются. И так далее, вплоть до индекса $m - 1$.

Число операций пропорционально $m * (n/m)^2 = n^2/m$, где m — любое (например, $n/10$). Метод не очень надежен: действительно, довольно часто элементы разбросаны по отрезку (ρ_{min}, ρ_{max}) неравномерно, а если большая часть элементов будет иметь один и тот же индекс i_0, то никакой экономии по времени не получится. Кроме того, сложно указать оптимальное m. Этот метод лучше не применять.

2.7. Быстрая сортировка («q-sort»)

Простейшая реализация алгоритма использует возможность рекурсивного вызова функции. Количество вызовов будет порядка $\log n$.

Алгоритм:

Определяется функция `qsort(a,n)` с двумя аргументами — указатель на начало сортируемого массива a и число элементов массива n.

При $n < 8$ (можно взять и $n < 16$) массив сортируется прямым упорядочением.

При $n \geqslant 8$

Переставляются элементы a_0 и $a_{n/2}$. Смысл этого объясним позже.

Элементы массива переставляются так, чтобы все элементы $a_i \leqslant a_0$ лежали левее, чем все элементы $a_i > a_0$, т. е. элемент a_0 используется как точка отсчета. Тем самым массив

делится на две части, граница между ними приходится на некоторый индекс m.

Далее, если это возможно, сдвигаем границу m в направлении к середине массива $n/2$.

Вызываем `qsort(a,m)` и `qsort(&(a[m]),n-m)`

(или `qsort(a+m,n-m)`, что есть синоним `qsort(&(a[m]),n-m)`).

Что касается переупорядочения элементов массива, то это можно делать по-разному:

Совсем просто:

Инициализируется индекс $i = 1$.

Далее, другой индекс j пробегает весь массив от 1 до $n-1$, и как только встречается элемент $a_j < a_0$, он меняется местами с элементом a_i, а индекс i увеличивается на единицу.

Окончательное значение индекса i и есть граница m.

Более изящно:

Инициализируются два индекса: i=1, $j = n-1$.

Далее повторяются такие действия:

Индекс i бежит вверх, пока не встретится элемент $a_i \geqslant a_0$ (при дополнительном условии $i \leqslant j$).

Индекс j бежит вниз, пока не встретится элемент $a_j \leqslant a_0$ (при дополнительном условии $j \geqslant i$).

Меняем местами a_i и a_j.

Повторяем эти действия до тех пор, пока индексы i и j не встретятся.

Граница m может лежать в точках $m = i = j$, $m = i-1$, $m = j+1$.

Как нетрудно понять, максимальная скорость достигается при делении массива приблизительно пополам (тогда число делений составит $\log n$). В действительности даже пропорция 2 : 1 вполне приемлема. А вот деление 10 : $(n-10)$ (или 20 : $(n-20)$) совершенно неприемлемо — если оно будет встречаться каждый раз, то число делений будет n, а число операций — n^2.

Формально все первоначальные расположения элементов равновероятны, но в реальной работе частично упорядоченные массивы встречаются чаще, чем неупорядоченные. Именно поэтому на первом этапе мы меняем a_0 с $a_{n/2}$, т. е. используем в качестве «точки отсчета» $a_{n/2}$, а не a_0 (которое вполне может оказаться меньше всех остальных элементов, так что массив разделится как $1 : n-1$).

Сдвиг границы m в направлении $n/2$ возможен, если при $m < n/2$ $a_{m+1} \leqslant a_0$ или при $m > n/2$ $a_{m-1} \geqslant a_0$. Если этого не сделать, то массив типа $(1, -1, 0, 0, 0, \ldots, 0)$ будет делиться как $1 : n-1$.

Число операций — в среднем $n \log n$, при невероятной неудаче — n^2. Это вообще в среднем самый быстрый метод, более того, его довольно просто реализовать. Метод вполне можно применять, если только возможная (хотя и очень маловероятная) ненадежность не фатальна.

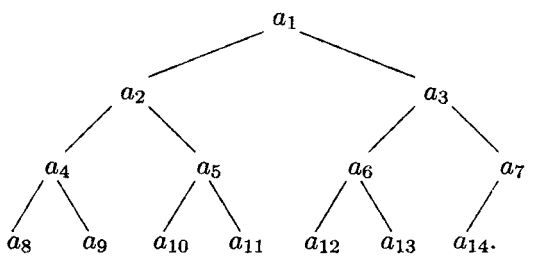
2.8. Метод двоичной кучи («heap-sort»)

Этот метод состоит из двух этапов.

1-й этап, создание кучи:

Кучей называется расположение элементов массива, при котором $a_i > a_{2i}$ и $a_i > a_{2i+1}$ для каждого i (если только индексы не вылезают за границы массива). Будем излагать алгоритм на примере массива из 14 элементов.

Графический образ кучи имеет вид



Черточки означают упорядоченность соответствующих элементов. Однако послойной упорядоченности в куче нет: вполне может быть, что $a_5 > a_3$.

Куча создается так:

Алгоритм:

Инициализируем индекс $i = 14/2$ и упорядочиваем тройку (a_i, a_{2i}, a_{2i+1}).

При этом, разумеется, умножение `2*i` надо писать как `i≪1`, а деление `i/2` — как `i≫1`.

Кроме того, сначала надо сравнить a_{2i} и a_{2i+1}, а уж потом больший из них с a_i.

Уменьшаем индекс i на единицу и повторяем то же самое.

При переходе на второй и более высокие слои (т. е. при переходе к a_3, a_2 и a_1) придется каждую выполненную перестановку сопровождать сравнением переставленного вниз элемента с двумя, оказавшимися под ним. Если он и на новом месте нарушает условие кучи, то его надо будет переставить еще ниже. Тем самым элемент a_2 вполне может «осыпаться» на самый нижний слой.

Когда мы дойдем до тройки (a_1, a_2, a_3), куча будет сформирована.

2-й этап, вытягивание кучи:

Алгоритм:

Запоминаем верхний элемент a_1 («вытягиваем его из кучи»).

Заполняем возникшую дырку одним из нижележащих элементов (a_2 или a_3) — тем, который больше. Новую дырку (например, a_3) также заполняем одним из нижележащих элементов (a_6 или a_7) — тем, который больше. И так далее, вплоть до нижнего слоя кучи. Дырку, возникающую в самом нижнем слое, заполняем элементом a_{14}, а на его место (14-е) помещаем a_1. Наконец, восстанавливаем порядок, нарушенный перемещением a_{14}, — его надо сравнить с новым вышележащим элементом и при необходимости переместить вверх. Если он переместился, то его снова надо сравнить с вышележащим, и т. д.

Далее повторяем все те же действия, только полагаем, что массив укоротился на единичку, т. е. состоит из 13 элементов. Вытягиваем из кучи текущий a_1, на его место вытягиваем a_2 или a_3 — тот,

который больше, и т. д. Дырку, возникающую в самом нижнем слое, заполняем элементом a_{13} а на его место помещаем a_1, и восстанавливаем порядок, нарушенный перемещением a_{13}.
И так далее, пока массив не укоротится до единичной длины.

Число операций всегда пропорционально $n \log n$, но, разумеется (как и для алгоритма q-sort), зависит от первоначального порядка элементов. В среднем этот алгоритм медленнее алгоритма q-sort в 2–3 раза. Кроме того, его чуть сложнее реализовать. Если у нас достаточно машинной памяти и мы можем позволить себе завести еще один массив того же размера, то алгоритм можно слегка укоротить: просто по очереди вытягивать элементы из кучи и заполнять ими другой массив, не занимаясь перестановками в конец исходного массива.

Этот алгоритм следует применять, если возможная ненадежность q-sort фатальна.

3. Арифметика произвольной точности

Иногда (хотя и довольно редко) оказывается, что значащих цифр в стандартном представлении действительных чисел для той или иной задачи не хватает. Следует понимать, что количество значащих цифр в действительном числе не есть свойство машины — это свойство компилятора. В языке «С» определены переменные типов `float`, `double` и `long double`, но не уточняется их размер. Как правило, `float` — это 4 байта, а `double` и `long double` — это 8 байт и ничем не отличаются друг от друга. Однако, в компиляторе BCC5 (Borland) переменная типа `long double` — это 10 байт. Разумеется, размер 8 байт привязан к параметрам математического сопроцессора, но распределение битов между мантиссой и показателем числа в разных компиляторах разное.

Опыт показывает, что переменными типа `float` лучше не пользоваться вовсе. Экономия памяти в 2 раза в действительности иллюзорна — как правило, памяти не хватает не для одномерных массивов, а для матриц, но размерность матрицы при замене `double` на `float` можно увеличить всего в $\sqrt{2} = 1.4$ раза, что вряд ли спасет положение. По скорости мы, как правило, только теряем, т. к. при операциях с `float`-числами они конвертируются в `double`, передаются сопроцессору, и затем результат конвертируется обратно во `float`. Между тем, утрата половины значащих цифр очень часто оказывается фатальной: очень многие алгоритмы, прекрасно

работающие с `double`-числами, наталкиваются на нехватку точности (накопление ошибок округления) при работе с `float`.

Итак, бывают задачи, в которых стандартных 8-байтовых `double` не хватает. В принципе, арифметика произвольной точности (и даже абсолютной точности, в виде рациональных чисел) реализована в системах аналитических вычислений (REDUCE, MAXIMA, Mathematica, Maple и т. п.). Однако они работают с числами как с аналитическими выражениями (хотя и состоящими из одного числа), т. е. очень медленно. Так что не следует вести численный счет с повышенным числом верных знаков на этих системах. Опыт показывает, что арифметика произвольной точности, реализованная на «С», может работать в 10^4 раз быстрее, чем арифметика в системах аналитических вычислений. Кроме того, некоторые встроенные численные функции Mathematic'и без предупреждения обрезают точность на 16 знаках, даже если заказанная точность — 50 знаков.

Итак, надо уметь самому писать арифметику произвольной точности. Изложим простейший алгоритм.

3.1. Представление

Число — одномерный массив типа «`int`». a_0 — это знак числа (± 1), a_1 — показатель степени, $a_2, a_3 \ldots$ — значащие цифры. Если Вам нужно 50 верных знаков, берите около 60 значащих цифр, т. е. с запасом в 10 знаков.

3.2. Сложение

Сложение проводится в точности как в начальной школе — столбиком.

Прежде всего следует выровнять порядки. Значащие цифры числа с меньшим порядком сдвигаются вправо на количество позиций, равное разнице порядков, а освободившееся место заполняется нулями.

При одноименных знаках чисел значащие цифры складываются. После этого справа налево (для всех i от $n-1$ до 3) проверяется переполнение значащих цифр. Именно, если $a_{i+1} > 10$, то a_i увеличивается на $a_{i+1}/10$, а в a_{i+1} остается остаток от деления на 10: $a_{i+1}\%10$. Наконец, при $x \equiv a_2 > 10$ все значащие цифры сдвигаются вправо на 1 позицию, порядок числа увеличивается на 1, а первым двум цифрам присваиваются значения: $a_2 = x/10$, $a_3 = x\%10$.

При разноименных знаках чисел на самом деле проводится вычитание. Выясняется большее из чисел (просто сравнением значащих цифр слева

направо, т. к. порядки уже выровнены). Далее из знаков большего числа вычитаются знаки меньшего числа. После этого справа налево (для всех i от $n-1$ до 3) проверяется отрицательность. Если $a_{i+1} < 0$, то a_i уменьшается на единичку, а к a_{i+1} добавляется 10. Наконец, если $a_2 = 0$ (отрицательным оно быть не может), то все значащие цифры сдвигаются влево на 1 позицию, позиция a_n заполняется нулем, а порядок уменьшается на 1. После этого вполне может снова оказаться $a_2 = 0$ (если мы вычитаем числа, у которых совпадает несколько старших значащих цифр). В этом случае мы вновь повторяем процедуру уменьшения порядка до тех пор, пока $a_2 \neq 0$. Если при сравнении чисел окажется, что они совпадают, то *ни в коем случае* не следует приравнивать ответ тождественному нулю. Надо оставить указание на точность, с которой этот ответ получен, т. е. считать, что в самой последней значащей цифре разности стоит не 0, а 1.

Вычитание:

Вычитание $a - b$ есть просто $a + (-b)$, т. е. смена знака у вычитаемого с последующим сложением.

3.3. Умножение

Умножение проводится в точности как в начальной школе — столбиком.

Знаки чисел умножаются, их порядки складываются, а значащие цифры произведения вычисляются по формуле

$$c_{k+2} = \sum_{l=0}^{k} a_{k-l+2} b_{l+2} \qquad k = 0 \ldots n-2.$$

После этого (как и в процедуре сложения) надо избавиться от переполнения в значащих цифрах. Это делается точно так же, единственное отличие заключается в том, что из младшего разряда a_{i+1} в старший разряд a_i вполне могут переноситься (добавляться) числа, превышающие 10. Поэтому не исключено, что на последнем этапе при $a_2 > 10$ процедуру увеличения порядка придется повторить несколько раз.

3.4. Деление

Вообще говоря, можно реализовать и деление уголком. Более того, при не очень большом количестве значащих цифр это может оказаться быстрее, чем предлагаемый алгоритм.

Однако в общем случае лучше все же использовать Ньютонов алгоритм: деление x/y — это умножение $x * \mathrm{sign}(y) * (1/|y|)$, а чтобы вычислить $1/|y|$, сооружаем последовательность:

$$x_{n+1} = x_n + x_n - x_n * x_n * |y|.$$

Сходимость к $1/|y|$ квадратичная, ибо если $x_n = 1/|y| + \epsilon$, то

$$x_{n+1} = 2/|y| + 2\epsilon - |y|(1/y^2 + 2\epsilon/|y| + \epsilon^2) = 1/|y| - |y|\epsilon^2,$$

т. е. число верных знаков на каждой итерации удваивается. Чтобы удачно начать последовательность, следует соорудить целое число из, например, первых 4 знаков (для числа $5.432 \cdot 10^{13}$ это будет 5432), поделить нацело $10\,000\,000/5432 = 1840$ и взять в качестве затравки x_1 число $1.840 \cdot 10^{-13-1}$.

Если мы взяли 10 запасных значащих цифр, то обрывать итерацию следует при совпадении x_n и x_{n+1} с точностью до 5 последних знаков. Дело в том, что последние значащие цифры не являются верными (они содержат ошибки округления). Поэтому если требовать точного совпадения x_n и x_{n+1}, то последовательность будет сходиться очень долго или вообще никогда не сойдется. Так что половина запасных верных знаков приходится на ошибки в делении.

3.5. Квадратный и другие корни

Извлечение квадратного корня можно реализовать с помощью последовательности[2]

$$x_{n+1} = 0.5 * (x_n + x_n + x_n - x_n * x_n * x_n * y).$$

Она сходится к $1/\sqrt{y}$, причем сходимость здесь также квадратичная. При $x_n = 1/\sqrt{y} + \epsilon$ имеем

$$x_{n+1} = (3/\sqrt{y} + 3\epsilon - y(1/y^{3/2} + 3\epsilon/y + O(\epsilon^2))/2 = 1/\sqrt{y} + O(\epsilon^2).$$

Совершенно аналогично, корень N-ой степени вычисляется с помощью последовательности

$$x_{n+1} = (1/N) * \left[(N+1)x_n - x_n^{N+1} * y\right],$$

которая сходится к $y^{-1/N}$ (сходимость опять-таки квадратичная).

[2] «Каноническая» последовательность $x_{n+1} = 0.5 * (x_n + y/x_n)$ в данном случае не очень подходит, т. к. в ней возникает деление.

3.6. Другие функции

- Функции, которые легкомысленно называют «элементарными», т. е. exp, sin, cos, ..., вполне можно реализовать просто как соответствующие ряды. При этом в случае больших аргументов следует упрощать себе жизнь: $\exp(2x) = [\exp(x)]^2$ (эту процедуру полезно повторить несколько раз); $\cos(x + 2\pi n) = \cos(x)$ и т. д.

- Функцию вывода числа на печать реализовать очень просто — достаточно напечатать все значащие цифры числа, снабдив вывод знаком, десятичной точкой, указателем порядка и самим порядком (что-нибудь вроде `-1.23456789012345678901234567890e-777`). При печати в строку (массив `char str[100];`) можно не печатать цифры в строку функцией `sprintf`, а просто писать присвоение: «`str[1]=a[2]+'0'; str[2]='.'; str[3]=a[3]+'0';`» и т. д.

- Опыт показывает, что понадобятся функции, которые конвертируют `double`-числа в наше представление и обратно.

Конечно, можно было бы просто вычислить `double`-эквивалент нашего представления по формуле

$$x = \mathrm{sign}(a_0) * 10^{a_1} * (a_2 + 0.1 * a_2 + 0.01 * a_3 + \ldots). \qquad (*)$$

Соответственно, обратное преобразование можно реализовать как выделение порядка `double`-числа (целая часть от $\log_{10} x$) и последовательное извлечение значащих цифр из мантиссы (выделяем целую часть (a_2), умножаем остаток на 10, выделяем целую часть (a_3), умножаем остаток на 10 и т. д.).

Но лучше упростить себе жизнь, воспользовавшись функциями:

```
sscanf(str,"%le",&x);
sprintf(str,"%23.15le",x);
```

Если напечатать наше число в строку `str` (при этом имеет смысл ограничиться 15 знаками и проверить, что порядок не слишком большой и не слишком маленький), то библиотечная функция `sscanf` прекрасно прочтет его оттуда в `double`-переменную `x`. Точно так же, если `double`-переменную `x` с помощью библиотечной функции `sprintf` напечатать в строку `str`, то она будет содержать практически в точности наше представление. (Опять же, значащие цифры можно извлекать просто

как: "`a[2]=str[1]-'0'; a[3]=str[3]-'0';`" и т. д.) Надо только помнить, что разные компиляторы даже фиксированный формат "`%23.15le`" выводят чуть-чуть по-разному (например, Watcom под DOS/Windows выводит показатель в 4 позиции (считая знак показателя), а gcc (GNU C) под Unix — в 3 позиции, и т. п.).

3.7. Пути улучшения алгоритма

Предложенный рецепт — это простейший вариант, пригодный в том случае, когда надо хоть как-то (пусть и очень неэффективно) реализовать арифметику произвольной точности. (Все равно она работает быстрее, чем в системах аналитических вычислений.)

Если хочется ускорить счет, то можно предпринять следующее:

Во-первых, использование представления по модулю 10 выбрано исключительно ради простоты ввода-вывода чисел в привычной нам десятичной системе. Совершенно ясно, что все изложенные алгоритмы можно реализовать по любому модулю, лишь бы не происходило переполнения в целочисленных операциях для каждого разряда (каждой значащей цифры). Компромиссным решением является переход от модуля 10 к модулю 10000. Переполнения при этом еще не произойдет (мы полагаем, что размер `int` — 4 байта), да и «десятичность» представления сохранится. Правда, необходимо помнить, что при таком модуле изменение «порядка» на 1 — это умножение (деление) на 10000. Переход от 10 к 10000 дает ускорение приблизительно в 7 раз.

Далее, при реализации арифметических операций с необходимостью возникают операции деления на модуль и взятия остатка при делении на модуль. Совершенно ясно, что и 10, и 10000 — совершенно неподходящие для этого числа. А вот модуль $2^{13} = 8192$ позволяет реализовать деление просто как побитовый сдвиг (`x/8192 = x≫13`), а взятие остатка — просто как побитовый «AND» (`x % 8192 = x & 8191`). Это позволяет ускорить вычисления приблизительно в 20 раз. Правда, за это ускорение придется заплатить: при конвертации чисел из нашего представления в `double` и обратно функции `sscanf` и `sprintf` станут бесполезными — конвертацию придется проводить с помощью соотношения, аналогичного (*), в котором 10 заменяется на 8192. Точно также придется реализовывать и вывод чисел на печать. Кроме того, надо помнить, что при работе по модулю 10 или 10000 конвертация числа (скажем) 0.1 из `double` в наше представление даст именно 0.10000000000000000000000000..., а вот по модулю

8192 получится 0.10000000000000000055511151...; чтобы получить именно 0.10000000000000000000000000..., надо создать числа 1 и 10 (это сделать легко) и разделить их друг на друга.

Во-вторых, имеет смысл все же реализовать деление уголком. Это довольно неприятная процедура, с проверками и возвратами (из-за возможного перебора при определении очередной значащей цифры), но некоторого ускорения добиться можно.

В-третьих, при реализации умножения массив значащих цифр ответа есть в точности свертка массивов, соответствующих сомножителям. Если свертку вычислять в лоб, то количество операций пропорционально n^2 (n — число значащих цифр). Однако если воспользоваться быстрым преобразованием Фурье (см. раздел «Быстрое преобразование Фурье»), то количество операций будет пропорционально $n \log n$. К сожалению, при той потребности в верных знаках, которая чаще всего встречается (не больше 50–100 десятичных знаков, что означает около 15–30 знаков по модулю 8192), быстрое преобразование Фурье еще не дает выигрыша по скорости. Так что этот способ ускорения эффективен только при необходимости иметь очень большое число верных знаков (несколько тысяч).

Наконец, уж совсем оптимальную реализацию можно получить, если всю эту арифметику писать на уровне ассемблера, т. е. фактически делать работу за тех, кто пишет компиляторы языка «С». Это возможно, но (как нам кажется) физик-теоретик не должен этим заниматься.

4. Случайные числа

4.1. Типовая структура генератора случайных чисел

Всякий генератор содержит:

- Функцию инициализации генератора

  ```
  void srand(unsigned long int),
  ```

  которая употребляется, во-первых, при отладке, чтобы получать каждый раз одну и ту же последовательность (например, можно поставить вызов "`srand(7);`"), во-вторых, при работе, чтобы при каждом запуске получать другую последовательность (обычный способ — вставить зависимость от физического времени: "`srand(time(&t));`").

- Функцию, выдающую очередное случайное значение

  ```
  unsigned int rand(void)
  ```

- И константу, определяющую диапазон значений генератора

  `RAND_MAX`.

  Если существенны хорошие корреляционные свойства генератора, ни в коем случае нельзя брать библиотечный генератор. В особенности если он имеет `RAND_MAX = 32767`. (Это всего как максимум 32767 разных значений, дальше они начнут повторяться, причем в точности в том же порядке.)

Библиотечный генератор — это, как правило, линейный генератор[3], который устроен так:

$$x_{n+1} = (a * x_n + b) \mod m.$$

Типичная реализация такова:

```
#define RAND_MAX 32767

unsigned long int n=1;

void srand(unsigned long int s) { n= s%32768L; }

int rand()
   {
   n=(4097*n+12345)%32768L;
   return (int)n;
   }.
```

Иными словами, следующее значение получается из предыдущего «почти линейным» преобразованием. Элемент «псевдослучайности» вносится операцией взятия остатка по модулю m. Разумеется, корреляционные свойства такой последовательности чисел ужасны. В особенности это относится к младшим битам чисел, т. к. младшие биты связаны гораздо «более линейным» преобразованием. Приведенный пример с модулем $32768=2^{15}$ является простейшей иллюстрацией этого утверждения, поскольку операция «mod 32768» — это просто отбрасывание старших битов, а младшие биты остаются нетронутыми. Отсюда следует, что уж если Вы пользуетесь библиотечным генератором, то не следует генерировать случайное целое число в диапазоне от 0 до $n-1$ как "`rand()%n`", следует написать "`rand() / ( (1+RAND_MAX)/n )`".

[3] Его еще называют *линейный конгруэнтный генератор*.

К сожалению, в большинстве библиотечных генераторов константы a, b и m подобраны не слишком тщательно. Это сказывается, во-первых, на периоде генератора (период всегда меньше модуля, но при неудачных a и b он может быть гораздо меньше модуля).

Во-вторых, это сказывается на свойствах всей последовательности как целого. Если по последовательности псевдослучайных чисел k_i, выдаваемых линейным генератором, построить набор точек на плоскости $\vec{x}_i = (k_{2i-1}, k_{2i})$, то, если бы не операция взятия модуля, все они лежали бы на одной прямой. Операция взятия модуля эквивалентна схлопыванию всех квадратов со стороной m, образующих квадратную сетку на плоскости, в один квадрат, начинающийся в точке $(0, 0)$. Поэтому эта одна прямая отобразится в один квадрат как множество параллельных и, возможно, совпадающих отрезков.

При неудачном выборе параметров a и b количество разных (несовпадающих) отрезков может оказаться очень малым, т. е. точки $\vec{x}_i$ окажутся не разбросанными случайным образом по квадрату, а лежащими на нескольких параллельных отрезках. Разумеется, заполнение этих отрезков не будет идти упорядоченно, каждая очередная точка будет появляться на том или ином месте того или другого отрезка, но когда наберется достаточное количество точек, «линейчатая структура» выявится очень отчетливо.

Кстати, очень большое количество несовпадающих отрезков также нежелательно, поскольку в алгоритме есть операция взятия модуля. Из-за этого построенные точки будут принадлежать не только набору отрезков с положительным наклоном K, но и набору отрезков с отрицательным наклоном $K - m$. Наши точки (а их не больше $2m$ штук) лежат на пересечениях этих двух наборов отрезков. Следовательно, большое количество отрезков с положительным наклоном означает малое количество отрезков с отрицательным наклоном, т. е. опять-таки отчетливую «линейчатую структуру». Ясно, что оптимальное количество отрезков, при котором точки заполняют квадрат хоть сколько-нибудь равномерно, — это $\sqrt{m}$.

Разумеется, это рассуждение можно повторить и для трехмерного пространства, четырехмерного пространства и т. д. При этом оптимальное количество гиперплоскостей, на которых разместятся все точки, будет всего $\sqrt[d]{m}$, где d — размерность пространства. Так что при m порядка 10000 в четырехмерном пространстве гиперплоскостей будет всего 10. Это еще одна причина, по которой генератор с маленьким m является совершенно неудовлетворительным.

К сожалению, сделать модуль очень большим невозможно. Если переменная `unsigned long` имеет размер 4 байта, то $a * m + b$ не должно превышать $2^{32} - 1$. Иначе будет происходить переполнение, т. е. к операции «mod m» добавится операция «mod 2^{32}», что ухудшает и так не очень хорошую последовательность. Сделать величину a маленькой и за счет этого увеличить m тоже нельзя.

Приведем примеры простейших генераторов, которые обладают периодом порядка модуля, имеют оптимальное количество гиперплоскостей и свободны от переполнения. Можно взять, например, такие параметры:

```
a1=7141 b1=54773 m1=259200;
a2=8121 b2=28411 m2=134456;
a3=4561 b3=51349 m3=243000.
```

Этими генераторами лучше не пользоваться, поскольку минимальные усилия позволяют получить гораздо более качественный генератор.

4.2. Простейший относительно удовлетворительный генератор

Переполнение, упомянутое выше, можно использовать, если подобрать удачные параметры a, b для модуля $m = 2^{32}$:

RAND_MAX=$2^{32} - 1$

```
n=n*1664525+1013904223.
```

В этом генераторе операция «mod m» происходит автоматически.

Конечно, у этого генератора не очень хороши корреляционные свойства, но он очень прост и у него сравнительно большой период (порядка 10^9). Так что если плохие корреляционные свойства не фатальны, им вполне можно пользоваться.

4.3. Улучшение корреляционных свойств

Основные недостатки линейных генераторов (плохие корреляционные свойства, и в особенности у младших битов, и небольшой модуль) довольно легко преодолеть.

Во-первых, можно перемешивать порядок, в котором выдаются числа. Для этого достаточно кроме основного генератора завести еще один, дополнительный генератор. Кроме того, надо создать буферный массив S произвольного, но достаточно большого размера n (n порядка нескольких сотен). При инициализации генератора буфер заполняется числами, которые выдает основной генератор. После этого при каждом очередном вызове

функции `rand()` первым делом вызывают дополнительный генератор и определяют случайный индекс k, лежащий в диапазоне $0 \ldots n-1$. Элемент массива S_k и будет возвращен в качестве ответа, а освободившееся (k-ое) место будет заполнено очередным числом, которое выдаст основной генератор. Этот рецепт существенно улучшает корреляционные свойства генератора.

Во-вторых, можно улучшить свойства младших битов, если завести не один, а два основных генератора. Если первый генератор выдает число n_1, а второй — n_2, то в качестве результата можно выдавать $n_1 + n_2/z$, где z порядка $\sqrt{m_2}$. Там самым старшие биты n_2 подмешиваются к младшим битам n_1. Впрочем, опыт показывает, что можно брать просто $|n_1 - n_2|$ или $(n_1 - n_2 + m_2) \bmod m_1$ — получается ничуть не хуже. Кстати, при наличии двух основных генераторов можно и не заводить дополнительный генератор (который определяет случайный индекс k). В качестве дополнительного можно использовать один из основных.

В-третьих, можно избежать переполнения даже при большом модуле. Действительно, взятие модуля означает вычитание $y * m$, где y — некоторое целое число. Допустим, что при делении m на a остаток равен r, а частное равно q. Тогда

$$a * x + b - y * m = a * x + b - y * (a * q + r) =$$
$$= a * (x - y * q) + b - y * r.$$

Следовательно, $y \approx x/q$. Итак, мы вычисляем: "`y=x/q`", затем вычисляем: "`z=a*(x-y*q)+b-y*r`", и затем требуем, чтобы z лежало в диапазоне от 0 до $m-1$ (из-за добавления b величина z может оказаться больше $m-1$, тогда надо вычесть m, а из-за вычитания $y * r$ величина z может оказаться отрицательной, тогда надо добавить m).

4.4. Совсем хороший линейный генератор

Это генератор, реализующий все идеи, изложенные в предыдущем пункте. Строятся два основных генератора, один из которых используется как дополнительный:

```
M1=2147483563 A1=40014 B1=12345 (Q1=53668 R1=12211);

M2=2147483399 A2=40692 B2=54321 (Q2=52774 R2=3791).
```

При инициализации буферный массив S длиной порядка 500 заполняется элементами $|n_1 - n_2|$, а дальнейшая генерация идет в точности как описано выше.

Этим генератором действительно можно пользоваться. Его период велик (порядка произведения $M1 * M2$), а корреляционные свойства не очень плохие.

4.5. Совершенно другой — разностный генератор

Свойства разностного генератора отличаются от свойств линейного генератора. Этот генератор тоже плох, но плох несколько по-другому. Так что если у Вас есть подозрение, что задача не считается из-за использования линейного генератора случайных чисел, можно проверить это путем перехода на разностный генератор.

Мы ограничимся изложением рецепта. Все разности, которые здесь встречаются, можно заменить на симметрические, т. е. на побитовую операцию XOR.

Раньше всего определяются константы

`M = 10^9` (это любое большое число)

`S = 161803398` (это специально подобранное число)

и заводится массив

```
long int tabl[56]
```

(будут использоваться индексы 1...55, причем число 55 менять нельзя).

Далее определяется операция, которая и вносит элемент псевдослучайности

```
long int op(x,y)
        { z=x-y; if(z<0) z+=M; return z; }.
```

Случайность связана с тем, что M то добавляется, то нет.

Инициализация генератора проводится так:

если s — это аргумент `srand`, то последнему элементу массива и двум `long int` переменным `k1` и `k2` присваиваются такие значения:

```
k1=tabl[55]=abs(S-s)% M;
k2=1;
```

После чего идет заполнение массива:

Для всех `i` от 1 до 54

```
i2=(21*i)%55;
tabl[i2]=k2;
k2=op(k1,k2);
k1=tabl[i2];
```

После чего идет дополнительное внесение псевдослучайности:

Не менее 5 раз надо повторить следующее:

Для всех i от 1 до 55

```
i2=1+(i+30)%55;
tabl[i]=op(tabl[i],tabl[i2]);
```

Наконец, инициализируются два счетчика

```
n1=0; n2=31;
```

При каждом вызове генератора происходит следующее:

Оба счетчика (n1 и n2) увеличиваются на единицу, причем значение 56 заменяется на 1.

После этого проводят выкладку

```
tabl[n1]=op(tabl[n1],tabl[n2]);.
```

Новое значение tabl[n1] возвращают как очередное случайное число.

Это довольно быстрый, но, разумеется, не очень качественный генератор. Его удобно использовать именно как альтернативу линейным генераторам.

4.6. Генерация псевдонепрерывных распределений

Если нужно создать равномерное распределение действительных чисел x_i на отрезке $[0,1]$, то, как ни странно, простейшая идея $x_i = n_i/$`(double)RAND_MAX` оказывается совершенно правильной. Как правило, не возникает необходимости сооружать конструкции типа $x_i = (n_{2i-1} + 10^{-7} n_{2i})/$`(double)RAND_MAX` (тут еще надо проверить, что x_i не превышает 1).

Пусть нужно построить распределение с плотностью вероятности $f(x)$ и функцией распределения $F(x) = \int_{-\infty}^{x} f(x)\,dx$, удовлетворяющей условию нормировки $F(\infty) = 1$. Тогда сначала генерируется равномерное распределение x_i, $i = 1, \ldots, N$ на отрезке $[0,1]$, а потом вычисляются $y_i = F^{-1}(x_i)$, где $F^{-1}(x)$ — функция, обратная к $F(x)$. Величины y_i распределены с плотностью вероятности $f(x)$. Действительно, число точек, попавших в отрезок δy, есть число точек, попавших в соответствующий отрезок $\delta x = F'(y)\delta y$, а это число есть попросту $N\delta x$, что дает для плотности вероятности $(N\delta x/N)/\delta y = F'(y) = f(y)$.

Если обратная функция F^{-1} не вычисляется или вычисляется с трудом (как, например, для Гауссового распределения), можно поступить иначе.

Если распределение $f(x)$ имеет компактный носитель ($f(x)$ отлично от нуля только на отрезке $(x_0, x_0 + \Delta x)$, то накроем это распределение прямоугольником $(x_0 < x < x_0 + \Delta x, 0 < y < \Delta y)$ (здесь $\Delta y = \max f(x)$) и набросаем в этот прямоугольник равномерно распределенные точки (x, y):

```
x = x0 + Δx * rand() / (double)RAND_MAX,
y = Δy * rand() / (double)RAND_MAX.
```

Все точки, у которых $y < f(x)$, принимаются, а все прочие — отбрасываются. Очевидно, что принятые точки распределены по x с плотностью вероятности $f(x)$.

Для распределения с некомпактным носителем, как правило, можно выделить отрезок длиной Δx, вне которой $f(x)$ практически равна нулю, и применить предыдущий рецепт.

Если такое выделение отрезка Δx (и выбрасывание всего остального диапазона x) нежелательно или невозможно, то распределение $f(x)$ можно накрыть не прямоугольником, а другим (причем ненормированным) распределением $g(x)$ ($G(\infty) = \int_{-\infty}^{\infty} g(x) \neq 1$), у которого обратная функция $G^{-1}(x)$ легко вычисляется (например, $g(x) = N/(x^2 + a^2)$). При этом для всех x должно выполняться $g(x) > f(x)$, что обычно легко обеспечить, домножая $g(x)$ на константу (и увеличивая тем самым $G(\infty)$). Набросаем под кривую $g(x)$ равномерно распределенные точки:

```
x = G⁻¹( G(∞) * rand() / (double)RAND_MAX ),
y = g(x) * rand() / (double)RAND_MAX.
```

Все точки, у которых $y < f(x)$, принимаются, а прочие — отбрасываются, так что принятые точки будут распределены с плотностью вероятности $f(x)$.

5. Интерполяция

Самая общая постановка задачи выглядит так: мы располагаем набором $y_1, \ldots, y_n$ — значениями функции $y(x)$ в точках $x_1, \ldots, x_n$. Надо так или иначе получить значение функции $y(x)$ в некоторой точке x. Мы не будем различать интерполяцию и экстраполяцию, более того, значение x может быть и комплексным (при действительных $x_1, \ldots, x_n$), т. е. вполне возможно численное «аналитическое продолжение» функций.

Как ни странно, в теоретической физике интерполяция встречается не реже, чем в экспериментальной. При табуляции спецфункции, которая вычисляется медленно и с трудом, есть смысл вычислить ее на достаточно редкой сетке, а в остальные точки проинтерполировать. Задача аналитического продолжения функции с действительной оси тоже встречается. Кроме того, интерполяция является составной частью многих других алгоритмов.

5.1. Полиномиальная интерполяция

Формула для полиномиальной интерполяции, разумеется, хорошо известна — это просто формула Лагранжа

$$P(x) = \sum_{i=1}^{n} y_i \prod_{k \neq i} \frac{x - x_k}{x_i - x_k}.$$

Однако было бы большой ошибкой пользоваться непосредственно этой формулой при реализации полиномиальной интерполяции.

Следует применять рекурсивную процедуру. Для каждой точки x, куда мы хотим проинтерполировать, надо проделать такие операции:

Алгоритм:

Сначала определяются величины (одномерный массив):

n величин: $\quad P_{1,1} \equiv y_1 \qquad P_{2,2} \equiv y_2 \qquad \ldots \qquad P_{n,n} \equiv y_n.$

Затем по формуле

$$P_{i,k} = \frac{P_{i,k-1}(x_k - x) + P_{i+1,k}(x - x_i)}{(x_k - x_i)} \qquad (*)$$

вычисляют последовательно

$n-1$ величину: $\quad P_{1,2} \qquad P_{2,3} \ldots \qquad P_{n-1,n}$

$n-2$ величины: $\quad P_{1,3} \qquad P_{2,4} \ldots \qquad P_{n-2,n}$

$\ldots$

две величины: $\quad P_{1,n-1} \qquad P_{2,n}$

наконец, одну величину: $\quad P_{1,n},$

которая и является искомым ответом.

формула $(*)$ устроена так, что $P_{i,k}$ является полиномом $(k-i)$-го порядка относительно x, который в точках $x = x_i, \ldots, x_k$ имеет правильные значения $y_i, \ldots, y_k$. Действительно, для $P_{k,k}$ это верно по определению, а при каждом последующем применении $(*)$ мы получаем, что:

- при $x = x_i$ мы получаем в точности $P_{i,k-1}$, равное y_i;
- при $x = x_k$ мы получаем в точности $P_{i+1,k}$, равное y_k;
- при остальных x_j как $P_{i,k-1}$, так и $P_{i+1,k}$ равны y_j, т. е. мы получаем y_j.

Разумеется, нет никакой необходимости помнить промежуточные $P_{i,k}$, так что при реализации алгоритма двумерный массив не нужен — достаточно одномерного.

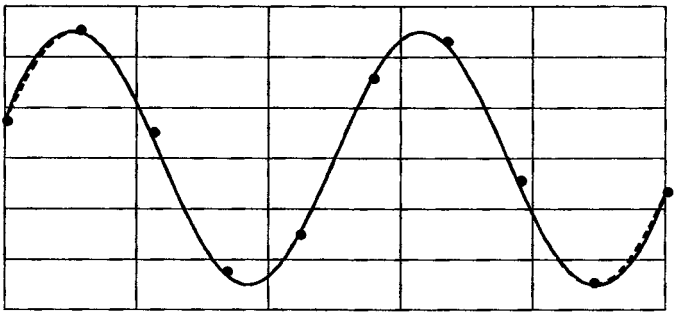
С одной стороны, при этой процедуре для каждого очередного x все выкладки приходится проводить заново. Но с другой стороны, эта рекурсивная процедура позволяет следить за ошибками. Действительно, ошибку можно оценить по последней поправке, т. е. по разнице между окончательным ответом $P_{1,n}$ (интерполяция по n точкам) и предыдущими двумя ответами $P_{1,n-1}$ и $P_{2,n}$ (интерполяции по $n-1$ точкам).

Следует только иметь в виду, что если у нас четное число точек и при этом функция y_k на этих точках четная, т. е. $y_{n-k+1} = y_k$, то последняя поправка тождественно равна нулю (просто из соображений четности оба предпоследних полинома совпадают между собой и совпадают с окончательным ответом). То же самое получается в случае нечетного количества точек при нечетной функции y_k ($y_{n-k+1} = -y_k$). Как ни странно, оба эти случая нередко встречаются в работе физика-теоретика. В этих случаях оценить ошибку можно по предпоследней поправке, т. е. сравнивая $P_{1,n}$ с $P_{1,n-2}$, $P_{2,n-1}$ и $P_{3,n}$.

Оценивать ошибку совершенно необходимо, иначе можно получить бессмысленный результат.

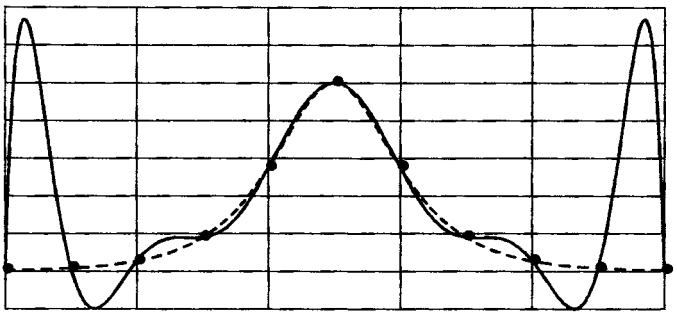
Ошибки при полиномиальной интерполяции, как правило, возрастают вблизи концов интервала, на котором лежат $x_1 \ldots x_n$. Кроме того, обычно с возрастанием количества точек n точность растет. Однако это верно только для тех функций, которые хорошо аппроксимируются полиномом, т. е. для функций с большим радиусом сходимости степенного ряда. Иными словами, если полюса функции лежат достаточно далеко от точек $x_1 \ldots x_n$, то чем больше количество точек n, тем точнее интерполяция. Вот пример

функции (что-то вроде синуса), для которой полиномиальная интерполяция достаточно хороша и увеличение n приводит к уменьшению ошибок:



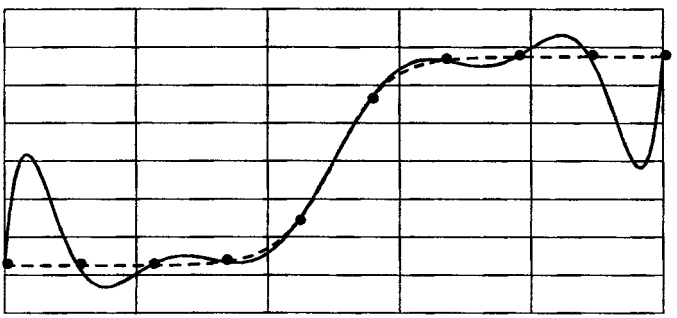
Сплошной линией показан результат интерполяции, а штриховой — исходная функция. Здесь интерполяция идет по 10 точкам; фактическая относительная погрешность интерполяции (разница между функцией и полиномом) меняется от 0.1% в середине интервала до 5% на его концах, и эти 5% заметны на приведенной картинке. Оценка погрешности, заложенная в алгоритме, дает даже несколько завышенное значение погрешности. Если интерполяцию этой же функции проводить по 20 точкам, то погрешность интерполяции уменьшится до 10^{-6}.

Однако если условие удаленности от полюсов не выполняется, то увеличение числа точек n не улучшает, а ухудшает ситуацию — простое соединение соседних точек прямыми (линейная интерполяция по двум соседним точкам) может дать более точный ответ, чем полиномиальная интерполяция по всем точкам. Типичная функция, для которой полиномиальная интерполяция не очень-то применима, это функция следующего вида:



Сплошной линией показан результат интерполяции, а штриховой — исходная функция; интерполяция идет по 11 точкам. Говорить о погрешности в данном случае бессмысленно. Эта функция представляет собой что-то вроде $1/(x^2+1)$, т. е. имеет полюса $x = \pm i$ вблизи точек x_i, поэтому радиус сходимости степенного ряда очень мал и результаты полиномиальной интерполяции неудовлетворительны. Легко проверить, что с ростом числа точек n отклонение от функции только нарастает; более того, предыдущий полином совершенно не похож на последующий. Поэтому оценка погрешности, заложенная в алгоритме, позволяет детектировать непригодность полиномиальной интерполяции для этой функции.

Другой аналогичный пример:



Эта функция представляет собой что-то вроде $\tanh(x)$, т. е. имеет полюса $x = \pm\pi i/2$ вблизи точек x_i, и полиномиальная интерполяция для нее не подходит. Опять-таки, это легко обнаружить, если оценивать погрешность интерполяции.

5.2. Рациональная интерполяция

При рациональной интерполяции строится дробно-рациональная функция

$$(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0)/(b_m x^m + b_{m-1}x^{m-1} + \ldots + b_0),$$

которая в точках $x_1, \ldots x_N$ равна $y_1, \ldots y_N$. Поскольку числитель и знаменатель определены с точностью до общего множителя, то количество точек, которые определяют эту функцию, будет $N = (m+1) + (n+1) - 1$.

Разумеется, с помощью рациональной интерполяции можно успешно проинтерполировать гораздо более широкий класс функций, чем с помощью

полиномиальной. В особенности это относится к задаче об аналитическом продолжении функций. Однако необходимо учитывать, что рациональная интерполяция способна давать сингулярности (в том числе и на действительной оси) в точках $x = x_0$, являющихся корнями знаменателя. В некоторых случаях (когда у функции действительно есть сингулярность) это является достоинством метода. К сожалению, бывает и так, что из-за погрешностей в начальных данных (y_k известны с недостаточной точностью), рациональная интерполяция дает ложные сингулярности на действительной оси (или почти на действительной оси, т. е. поведение типа $1/((x-a)^2+\varepsilon^2)$ или $(x-a)/((x-a)^2+\varepsilon^2)$ при достаточно малом ε), в то время как сама функция в окрестности точки $x = a$ подобного поведения не имеет.

При данном количестве точек фиксирована только сумма $m + n$, поэтому мы можем строить разные рациональные функции — или «более полиномиальные» ($n > m$), или «обратные полиномиальные» ($n < m$). Ясно, что есть функции, для которых предпочтителен первый вариант, и есть функции, для которых предпочтителен второй. Если никакой априорной информации о функции нет, то в среднем оптимальной будет «балансированная» рациональная функция ($n = m$ или $n = m \pm 1$).

Существует по крайней мере три разных алгоритма для рациональной интерполяции.

Метод прямого вычисления:

Полагаем $b_0 \equiv 1$ и пишем систему линейных уравнений $(m+n+1)\times$ $\times(m+n+1)$ на коэффициенты $a_0, \ldots, a_n$ и $b_1, \ldots, b_m$:

$$(a_n x_k^n + a_{n-1}x_k^{n-1} + \ldots + a_0) - y_k(b_m x_k^m + b_{m-1}x_k^{m-1} + \ldots + b_1 x_k^1) = y_k;$$

$k = 1, \ldots n+m+1$. Эта система легко решается (см. раздел «Системы линейных уравнений»).

С одной стороны, этот алгоритм прост и нагляден; он позволяет брать произвольную комбинацию m и n; он позволяет получить явный вид функции; для фиксированного набора данных $x_1 \ldots x_N$, $y_1 \ldots y_N$ коэффициенты $a_0 \ldots a_n$ и $b_1 \ldots b_m$ находятся один раз и навсегда, а дальше можно интерполировать в неограниченное количество точек x — достаточно просто подставить x в готовую формулу.

С другой стороны, в этом алгоритме не заложен контроль точности; в нем довольно велик риск заработать большие ошибки округления (в системе

линейных уравнений может быть очень большой перепад между коэффициентами — например, если $x_1 = 1$, $x_2 = 10$, а m и n порядка 10).

В общем, лучше этот метод не применять.

Рекурсия переворотом дроби:

Строим рекурсию:

Рациональная функция $R^{(n)}(x)$ должна равняться $y_k \equiv y_k^{(n)}$ в точках x_k, $k = 1, \ldots, n$.

Определяем рациональную функцию $R^{(n-1)}(x)$ соотношением

$$R^{(n)}(x) = (x - x_n)/R^{(n-1)}(x) + y_n^{(n)},$$

тогда $R^{(n-1)}(x)$ должна равняться $y_k^{(n-1)}$ в точках x_k, $k = 1, \ldots, n-1$, где

$$y_k^{(n-1)} \equiv (x_k - x_n)/(y_k^{(n)} - y_n^{(n)}).$$

Далее, определяем рациональную функцию $R^{(n-2)}(x)$ соотношением

$$R^{(n-1)}(x) = (x - x_{n-1})/R^{(n-2)}(x) + y_{n-1}^{(n-1)},$$

тогда $R^{(n-2)}(x)$ должна равняться $y_k^{(n-2)}$ в точках x_k, $k = 1, \ldots, n-2$, где

$$y_k^{(n-2)} \equiv (x_k - x_{n-1})/(y_k^{(n-1)} - y_{n-1}^{(n-1)}).$$

И так далее. Наконец,

$$R^{(2)}(x) = (x - x_2)/R^{(1)}(x) + y_2^{(2)},$$

тогда $R^{(1)}(x)$ должна равняться $y_1^{(1)}$ в точке x_1, где

$$y_1^{(1)} \equiv (x_1 - x_2)/(y_1^{(2)} - y_2^{(2)}).$$

Следовательно, $R^{(1)}(x)$ — это просто число $y_1^{(1)}$.

На прямом ходе рекурсии мы вычисляем сначала все $y_k^{(n-1)}$, потом все $y_k^{(n-2)}$, и т. д., вплоть до $y_1^{(1)}$. Этот этап никак не зависит от той точки x, куда мы будем интерполировать.

На обратном ходе рекурсии при интерполяции в некоторую точку x мы вычисляем сначала $R^{(1)}(x) = y_1^{(1)}$, затем $R^{(2)}(x)$, затем $R^{(3)}(x)$, и так далее, вплоть до искомого ответа $R^{(n)}(x)$. Заметим, что на этом обратном ходе рекурсии помнить надо только «диагональные» значения $y_k^{(k)}$ ($k = 1, \ldots, n$), т. е. двумерный массив для этого алгоритма заводить не нужно.

Этот метод выдает балансированную дробно-рациональную функцию. Впрочем, его легко модифицировать так, чтобы степень числителя отличалась в ту или иную сторону от степени знаменателя (достаточно не переворачивать дробь, т. е. не делить, а умножать в одном или нескольких шагах рекурсии).

К сожалению, несмотря на рекурсивный характер алгоритма, предыдущие ответы $R^{(k-1)}(x)$ не есть приближение к последующим ответам $R^{(k)}(x)$. Напротив, алгоритм таков, что $R^{(k)}(x)$ прыгают от шага к шагу (... 0.001, 1000.0, 0.0007, 1500.0 ...), т. е. мы не имеем возможность контролировать точность интерполяции.

Этот метод имеет все недостатки предыдущего, но не обладает его достоинствами, так что лучше его не применять.

Рекурсия, аналогичная полиномиальной:

Алгоритм:

В точности как при полиномиальной интерполяции, для каждой точки x (в которую мы хотим интерполировать) сначала определяются два набора величин (два одномерных массива):

n величин: $R_{1,0} \equiv 0 \qquad R_{2,1} \equiv 0 \qquad \ldots \qquad R_{n,n-1} \equiv 0$

и

n величин: $R_{1,1} \equiv y_1 \qquad R_{2,2} \equiv y_2 \qquad \ldots \qquad R_{n,n} \equiv y_n.$

После этого с помощью формулы

$$R_{i,k} = R_{i+1,k} + \frac{R_{i+1,k} - R_{i,k-1}}{\left(\dfrac{x - x_i}{x - x_k}\right)\left(1 - \dfrac{R_{i+1,k} - R_{i,k-1}}{R_{i+1,k} - R_{i+1,k-1}}\right) - 1} \qquad (**)$$

последовательно вычисляются:

n величин: $R_{1,2} \qquad R_{2,3} \qquad \ldots \qquad R_{n-1,n}$

$n-1$ величина: $R_{1,3} \qquad R_{2,4} \qquad \ldots \qquad R_{n-2,n}$

...

две величины: $R_{1,n-1} \qquad R_{2,n}$

наконец, одна величина: $R_{1,n},$

которая и является искомым ответом.

Очень легко проверить, что дробно-рациональная функция $R_{i,k}$ обладает в точности теми же свойствами, что и полином $P_{i,k}$, т. е. в точках

$x = x_i, \ldots, x_k$ имеет правильные значения $y_i, \ldots, y_k$. Несколько сложнее убедиться, что на каждом этапе рекурсии происходит не одновременное, а поочередное увеличение на единицу степеней полиномов, стоящих в числителе и знаменателе дробно-рациональных функций.

Разумеется, все деления, входящие в формулу (**), должны сопровождаться проверками на нуль в знаменателе (например, при $x = x_k$ всю большую дробь следует занулить).

Как и для полиномиальной интерполяции, алгоритм позволяет оценить погрешность по последней поправке (разнице между $R_{1,n}$ и $R_{1,n-1}$ или $R_{2,n}$).

Существует одна малоизвестная возможность сбоя этого алгоритма: он гибнет, если вдруг одно или несколько y_i обращаются в нуль (или оказываются машинным нулем по сравнению с другими y_i). Это связано с тем, что на первом обороте алгоритма мы имеем функцию $1/(ax + b)$, которая не может в одной точке равняться нулю, а в другой точке быть отличной от нуля. Этот недостаток алгоритма легко исправить путем сдвига всех y_k на произвольную константу C до интерполяции и сдвига полученного ответа обратно на $-C$ после интерполяции.

Изложенный метод позволяет оценить погрешность интерполяции, так что его вполне можно использовать.

5.3. Фурье-интерполяция

При Фурье-интерполяции сетка (набор значений x_i) не произвольна — она должна быть равномерна ($x_i = a + dx * i$). Ясно, что любую равномерную сетку линейным преобразованием можно превратить в сетку $x_i = i$. Выпишем дискретное преобразование Фурье для набора значений y_i и обратное преобразование Фурье:

$$c_k = \sum_0^{N-1} y_n \exp(2\pi i k n/N)/\sqrt{N};$$

$$y_n = \sum_0^{N-1} c_k \exp(-2\pi i k n/N)/\sqrt{N}.$$

Тогда для любой промежуточной точки x можно написать:

$$f(x) \simeq \sum_0^{N-1} c_k \exp(-2\pi i k x/N)/\sqrt{N}.$$

Ниже будет показано (см. раздел «Быстрое преобразование Фурье»), что вычисление c_k — это не n^2 операций, а $n \log n$ операций. При вычислении $f(x)$ для каждого данного x надо вычислить весь набор значений $\exp(-2\pi ikx/N)$ (т. е. $\sin(2\pi kx/N)$ и $\cos(2\pi kx/N)$) при $k = 0, \ldots, N-1$. Однако, разумеется, тригонометрические функции надо считать только один раз — для $k = 1$, для остальных k следует использовать рекурсию.

При Фурье-интерполяции довольно часто имеет смысл отклониться от процедуры интерполяции в точном смысле этого слова. Именно, довольно часто данные y_i известны с некоторой погрешностью. Для физика-теоретика основной источник погрешностей в данных — это ошибки округления и принципиальная неточность алгоритма, который применяется при вычислении y_i. Так вот, если в тех значениях y_i, которыми мы располагаем, есть погрешности, то, с точки зрения Фурье-анализа, они, скорее всего, представляют собой высокочастотный шум. В этом случае стоит отклониться от точной интерполяции и подавить этот шум. При этом в точках x_i интерполяция будет попадать не совсем в y_i, но функция $y(x)$ будет более гладкая.

Где именно лежит граница между высокочастотными компонентами самой функции $y(x)$ и шумом, разумеется, зависит от конкретной задачи. Как правило, шумом можно считать диапазон значений k от $N/4$ до $3N/4$ или $N/6$ до $5N/6$. Однако *ни в коем случае не следует* просто занулять c_k в этой области. Дело в том, что такое действие означает умножение Фурье-образа функции c_k на θ-функцию. При этом сама функция y_n свертывается с Фурье-образом θ-функции, т. е. свертывается с быстро осциллирующей и медленно убывающей функцией. Такая операция не имеет ничего общего со сглаживанием высокочастотного шума.

Надо умножить c_k на что-то, напоминающее θ-функцию, но не имеющее скачков, например, на

$$1.0 - 0.5 \tanh(\lambda(k - N/6)) + 0.5 \tanh(\lambda(k - 5N/6))$$

Это гладкая функция, ее Фурье-образ не осциллирует и достаточно быстро убывает, но при этом она равна 1 в окрестностях $k = 0$ и $k = N$ (низкие частоты) и равна 0 в области $N/6 < k < 5N/6$ (высокочастотный шум).

5.4. Чебышевская интерполяция

Полиномы Чебышева 1-го рода определяются так:

$$T_n(x) = \cos(n \arccos(x)),$$

Вычисляются они, разумеется, рекурсией:

$$T_{n\pm1}(x) = \cos(n\arccos(x))\cos(\arccos(x)) \mp$$

$$\mp \sin(n\arccos(x))\sin(\arccos(x)),$$

откуда

$$T_{n-1}(x) + T_{n+1}(x) = 2xT_n(x),$$

причем

$$T_0(x) = 1 \qquad T_1(x) = x.$$

Корни n-го полинома есть, очевидно, $x_k = \cos(\pi(k+1/2)/n), k = 0, \ldots n-1$. Легко проверить, что значения полиномов T_l при $l = 0, \ldots n-1$ в точках x_k образуют базис:

$$e_{lk} \equiv T_l(x_k) = \cos\left(\frac{\pi l}{n}(k+1/2)\right),$$

т. е.

$$\sum_{k=0}^{n-1} e_{lk}e_{l'k} = \delta_{ll'}M_l, \qquad \text{где} \quad M_l = (1+\delta_{l0})\,\frac{n}{2},$$

$$\sum_{l=0}^{n-1} e_{lk}e_{lk'} = \frac{1}{2} + \delta_{kk'}\frac{n}{2}.$$

Действительно, поскольку

$$\sum_{k=0}^{n-1} \cos\left(\frac{\pi m}{n}(k+1/2)\right)$$

отлично от нуля только при $m = 0$ (для четных m это сумма по полному периоду, а для нечетных m замена $k \to n-1-k$ меняет знак суммы на противоположный). Поэтому

$$\sum_{k=0}^{n-1} \cos\left(\frac{\pi l}{n}(k+1/2)\right)\cos\left(\frac{\pi l'}{n}(k+1/2)\right) =$$

$$= \sum_{l=0}^{n-1} \frac{1}{2}\cos\left(\frac{\pi(l-l')}{n}(k+1/2)\right) + \frac{1}{2}\cos\left(\frac{\pi(l+l')}{n}(k+1/2)\right) =$$

$$= \delta_{ll'}(1+\delta_{l0})\frac{n}{2}.$$

Далее, выражение

$$\sum_{l=0}^{n-1} \cos\left(\frac{\pi lk}{n}\right)$$

при $k = 0$ равно n; при четных $k \neq 0$ равно нулю (это сумма по периоду); при нечетных k равно первому слагаемому, т. е. единице (поскольку сумма $\sum_{l=1}^{n-1} \cos(\pi lk/n))$ при замене $l \to n - l$ меняет знак на противоположный). Поэтому

$$\sum_{l=0}^{n-1} \cos\left(\frac{\pi l}{n}(k+1/2)\right) \cos\left(\frac{\pi l}{n}(k'+1/2)\right) = \sum_{l=0}^{n-1} \frac{1}{2} \cos\left(\frac{\pi l}{n}(k-k')\right) +$$

$$+ \frac{1}{2} \cos\left(\frac{\pi l}{n}(k'+k+1)\right) = \frac{1}{2} + \delta_{kk'} \frac{n}{2}.$$

Тем самым, если известны значения некоторой функции в точках x_k, то можно определить коэффициенты

$$c_l = \sum_{k=0}^{n-1} f(x_k) T_l(x_k)/M_l$$

и построить полином

$$T(x) = \sum_{l=0}^{n-1} c_l T_l(x),$$

который в точках $x = x_k$ будет равен $f(x_k)$.

На первый взгляд, раз интерполяция идет по полиномам, то это просто частный случай полиномиальной интерполяции для некоторого специального набора x_k.

Однако оказывается, что полином, полученный при помощи именно этого набора точек (корни x_k сгущаются на концах интервала), очень близок к минимаксному (тому, который минимизирует максимальное отклонение от функции, т. е. близок к ней по норме L^1). Кроме того, мы можем не просто вычислить полином в любой точке x — мы знаем еще и коэффициенты разложения c_k, что позволяет нам интегрировать и дифференцировать интерполируемую функцию.

Действительно, поскольку

$$T_n'(x) = -n \sin(n \arccos(x)) \frac{(-1)}{\sqrt{1-x^2}},$$

то

$$T'_{n\pm1}(x)/(n\pm1) = \Big[\sin(n\arccos(x))\cos(\arccos(x))\pm \\ \pm\cos(n\arccos(x))\sin(\arccos(x))\Big]\frac{1}{\sqrt{1-x^2}},$$

следовательно,

$$T'_{n+1}(x)/(n+1) - T'_{n-1}(x)/(n-1) = \\ = 2\cos(n\arccos(x))\sin(\arccos(x))/\sqrt{1-x^2} = 2T_n(x),$$

тем самым,

$$\sum_{l=0}^{n-1} c_l T_l(x) = \sum_{l=0}^{n-1} c_l[T'_{l+1}(x)/(l+1) - T'_{l-1}(x)/(l-1)]/2 = \\ = \sum_{l=0}^{n-1}(c_{l-1}-c_{l+1})T'_l/(2l) = \sum_{l=0}^{n-1} d_l T'_l.$$

Так что если

$$f(x) \approx \sum_{l=0}^{n-1} c_l T_l(x),$$

то

$$\int f(x)\,dx \approx \sum_{l=0}^{n-1} d_l T_l(x),$$

где $d_0 = const$ (постоянная интегрирования) и $d_l = (c_{l-1} - c_{l+1})/(2l)$ при $l \geqslant 1$.

И наоборот, если

$$f(x) \approx \sum_{l=0}^{n-1} d_l T_l(x),$$

то

$$f'(x) \approx \sum_{l=0}^{n-1} c_l T_l(x),$$

где коэффициенты c_l определяются рекурсивно (сверху вниз): $c_l = 2(l+1)d_{l+1} + c_{l+2}$. Чтобы начать рекурсию, нам придется допустить, что $c_{n-1} = 0$ и $c_{n-2} = 2(n-1)d_{n-1}$. (Раз мы располагаем только коэффициентами $d_0 \ldots d_{n-1}$ для функции $f(x)$, то узнать коэффициент c_{n-1} для производной $f'(x)$ мы все равно не можем, он зависит от d_n.)

5.5. Другие системы КОП

В принципе, функцию можно раскладывать и по другим системам классических ортогональных полиномов (КОП) — по Лежандрам, Лягеррам, Эрмитам и т. д. Но при этом трудновато найти точную формулу для коэффициентов. Один из методов поиска коэффициентов будет изложен в разделе «Численное интегрирование — метод Гаусса».

Компромиссное решение заключается в том, что можно найти приближенные значения коэффициентов — просто аппроксимируя интеграл, входящий в точную формулу для коэффициента

$$c_k = \int_a^b dx\; y(x) P_k(x) \rho(x)$$

соответствующей интегральной суммой по узлам сетки. К сожалению, это требует частой сетки и большого количества точек. Обязательной проверкой при использовании такого метода является обратная сборка функции в точках x_i — коэффициенты c_k определены правильно, если отклонение $\sum_k c_k P_k(x_i)$ от y_i невелико.

5.6. Сплайны

В принципе, существуют сплайны всех степеней, но использовать, пожалуй, стоит только кубические сплайны. Идея состоит в том, что вместо одной функции, проходящей через все точки, на каждом отрезке берется своя функция — кубическая парабола. Все эти кубические параболы на границах отрезка сшиваются, т. е. обеспечивается непрерывность функции, ее производной и ее второй производной. Кубическая парабола, попадающая в нужные точки на концах интервала (x_a, x_b), определяется двумя параметрами:

$$y(x) = [y_a(x - x_b) + y_b(x_a - x)]/(x_a - x_b) + \\ + A(x - x_a)^2(x - x_b) + B(x - x_a)(x - x_b)^2,$$

при этом

$$y'(x) = (y_a - y_b)/(x_a - x_b) + 2A(x - x_a)(x - x_b) + \\ + A(x - x_a)^2 + 2B(x - x_a)(x - x_b) + B(x - x_b)^2$$

$$y''(x) = (4A + 2B)(x - x_a) + (2A + 4B)(x - x_b).$$

Следовательно,

$$y'_a = (y_a - y_b)/(x_a - x_b) + B(x_a - x_b)^2,$$
$$y'_b = (y_a - y_b)/(x_a - x_b) + A(x_a - x_b)^2,$$
$$y''_a = (2A + 4B)(x_a - x_b) \qquad y''_b = (4A + 2B)(x_b - x_a),$$

т. е.

$$A = (y''_a + 2y''_b)/[6(x_b - x_a)] \qquad B = (2y''_a + y''_b)/[6(x_a - x_b)].$$

Следовательно, на сетке $x_1, x_2, \ldots, , x_N$ мы имеем N свободных параметров $y''_1, y''_2, , \ldots, y''_N$. На них пишутся $N - 2$ условия непрерывности первых производных $y'_2, y'_3, \ldots, y'_{N-1}$ на границах последовательных отрезков, т. е. производная $y'_{a:(n,n+1)}$ на левом конце отрезка $[x_n, x_{n+1}]$ должна совпадать с производной $y'_{b:(n-1,n)}$ на правом конце отрезка $[x_{n-1}, x_n]$:

$$\begin{aligned} y'_{a:(n,n+1)} =& (y_{n+1} - y_n)/(x_{n+1} - x_n) + \\ &+ (x_{n+1} - x_n)^2(2y''_n + y''_{n+1})/[6(x_n - x_{n+1})] = \\ =& (y_n - y_{n-1})/(x_n - x_{n-1}) + \\ &+ (x_n - x_{n-1})^2(2y''_{n-1} + y''_n)/[6(x_{n-1} - x_n)] = y'_{b:(n-1,n)}, \end{aligned} \qquad (*)$$

$n = 2, \ldots, N - 1$. Надо как-то фиксировать y''_1 и y''_N. Бывает, что известны значения концевых производных y'_1 и y'_N. Тогда это два искомых уравнения. Если производные неизвестны, то просто полагают $y''_1 = 0 = y''_N$ («натуральный сплайн»). Почему есть смысл ограничиться кубическим сплайном (ясно, что можно построить и сплайн более высокого порядка)? Потому что система уравнений для параметров сплайна $(*)$ будет тридиагональна, а такая система очень просто решается за N операций (см. раздел «Системы линейных уравнений»).

Как правило, интерполяция сплайном дает «самый гладкий» и «наименее извилистый» результат. Именно поэтому большинство графических программ, рисующих графики по точкам, используют именно сплайн. Однако особого математического смысла в сплайновой кривой нет, это просто способ провести нечто «визуально-гладкое» через данный набор точек.

5.7. Двумерная интерполяция: последовательная

Если функция двух аргументов $f(x, y)$ известна в узлах двумерной прямоугольной сетки, то ее легко проинтерполировать в любую точку (x, y). Для этого достаточно воспользоваться любым из изложенных одномерных методов или даже их гибридом. Действительно, при наличии сетки $(x_1, \ldots, x_N) \times (y_1, \ldots, y_M)$ мы сначала интерполируем по аргументу x в M точек $(x, y_1), , \ldots, (x, y_M)$. По полученным значениям функции в этих точках мы интерполируем по аргументу y и получаем значение в точке (x, y). При этом при интерполяции по аргументу x можно пользоваться одним одномерным методом, а при интерполяции по y — другим.

Естественный способ проверки правильности полученного ответа — проинтерполировать точно так же, но в другом порядке. Сначала по аргументу y в N точек $(x_1, y), \ldots, (x_N, y)$, а уж потом по полученным значениям функции в этих точках интерполировать по аргументу y в точку (x, y). Разница двух полученных ответов и дает оценку погрешности.

5.8. Двумерная интерполяция: билинейная и бикубическая

Это интерполяция типа сплайновой — т. е. в пределах каждого данного прямоугольника двумерной сетки строится своя интерполирующая функция.

Билинейная интерполяция:

Это линейная интерполяция по каждой из переменных (x и y) по отдельности, т. е. $F(x, y) = a + bx + cy + dxy$ (ясно, что «настоящей» двумерной линейной интерполяции быть не может — плоскость через произвольные 4 точки провести невозможно). Коэффициенты подбираются так, чтобы $F(x, y)$ совпадала с исходной функцией $f(x, y)$ в вершинах прямоугольника:

$$F(x, y) = \Big[g(y)(x - x_a) + h(y)(x_b - x)\Big]/(x_b - x_a),$$

где

$$g(y) = \Big[f(x_b, y_b)(y - y_a) + f(x_b, y_a)(y_b - y)\Big]/(y_b - y_a),$$
$$h(y) = \Big[f(x_a, y_b)(y - y_a) + f(x_a, y_a)(y_b - y)\Big]/(y_b - y_a).$$

Бикубическая интерполяция:

Это интерполяция функцией $F(x,y)$, являющейся полиномом третьей степени, по каждой из переменных по отдельности. Функция определяется с помощью 16-ти коэффициентов:

$$F(x,y) = \sum_{i,j=0}^{3} c_{ij} x^i y^j.$$

Условий, позволяющих найти c_{ij}, тоже должно быть 16, и самый естественный выбор — потребовать, чтобы сама F, обе ее первые производные $\partial_x F$, $\partial_y F$ и смешанная производная $\partial_x \partial_y F$ совпадали с f, $\partial_x f$, $\partial_y f$, $\partial_x \partial_y f$ во всех 4-х вершинах прямоугольника. При этом предполагается, что сетка равномерна, и производные исходной функции f вычисляются просто как разностные производные на сетке, например:

$$\partial_x f(x_n, y_n) = [f(x_{n+1}, y_n) - f(x_{n-1}, y_n)]/(2\Delta x).$$

Аналогично вычисляется и смешанная производная:

$$\begin{aligned}\partial_x \partial_y f(x_n, y_n) =& [\partial_y f(x_{n+1}, y_n) - \partial_y f(x_{n-1}, y_n)]/(2\Delta x) = \\ =& [\partial_x f(x_n, y_{n+1}) - \partial_x f(x_n, y_{n-1})]/(2\Delta y).\end{aligned}$$

Накладывая 16 условий на 16 коэффициентов c_{ij}, мы получим систему линейных уравнений 16×16, которая легко решается (см. раздел «Системы линейных уравнений»).

6. Поиск одномерных корней

Поиск одномерных корней (корней функций одного аргумента) является довольно простой задачей. Дело в том, что одномерный корень локализуем. Именно, если знак функции на концах некоторого отрезка разный, то на отрезке лежит один корень (или нечетное число корней).

Большинство методов, изложенных ниже, предполагают, что корень уже локализован на некотором отрезке, и надо только установить с заказанной точностью его положение.

Реальная задача, как правило, требует найти все корни на некотором отрезке. Как ни странно, примитивная табуляция функции на отрезке с достаточно мелким шагом остается основным методом предварительной

локализации корней на отрезке. Разумеется, желательно проверить, что мы локализовали (разделили) все корни: если при уменьшении шага (скажем, в 10 раз) количество найденных корней не возрастает, то есть надежда, что мы локализовали все корни.

6.1. Метод деления пополам

Пусть корень локализован на отрезке $[a, b]$.

Алгоритм:

Вычисляем $y_a = f(a)$, $y_b = f(b)$ и проверяем, что знаки y_a и y_b действительно разные.

Повторяем следующие действия:

Вычисляем $c = (a + b)/2$, $y_c = f(c)$.

Если знаки y_a и y_c одинаковы, то корень лежит на отрезке $[c, b]$, объявляем точку c новой точкой a: $a = c$, $y_a = y_c$.

В противном случае корень лежит на отрезке $[a, c]$, объявляем точку c новой точкой b: $b = c$, $y_b = y_c$.

Если длина нового отрезка $|a - b|$ меньше заказанной точности определения корня, то завершаем работу и возвращаем в качестве ответа $(a + b)/2$.

Метод обладает линейной сходимостью, т. е. число найденных верных знаков растет линейно с количеством операций n (погрешность, т. е. длина интервала $|a - b|$, ведет себя как 2^{-n}).

Если скорость нахождения корня не очень критична (т. е. если корни ищутся один раз и навсегда, а не на каждом обороте какого-нибудь итерационного алгоритма), то метод вполне можно применять.

6.2. Линейная интерполяция без проверки знаков

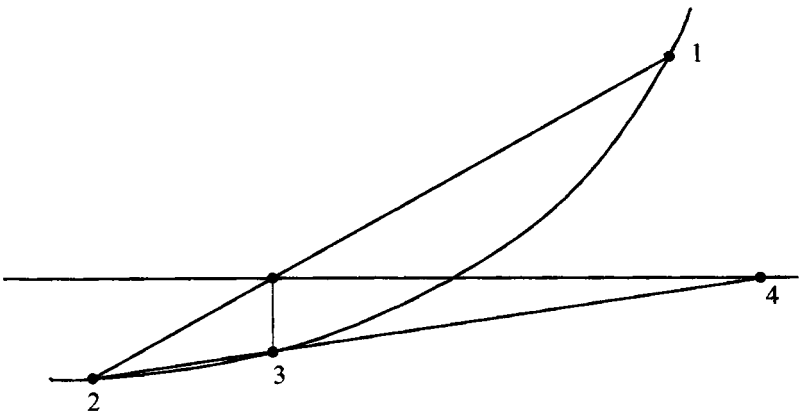
По двум точкам (двум предыдущим приближениям к корню) (x_{n-1}, y_{n-1}) и (x_n, y_n) строят прямую

$$x = [x_{n-1}(y_n - y) + x_n(y - y_{n-1})]/[y_n - y_{n-1}]$$

и находят ее корень:

$$x_{n+1} = [x_{n-1}y_n + x_n y_{n-1}]/[y_n - y_{n-1}] \qquad y_{n+1} = f(x_{n+1}).$$

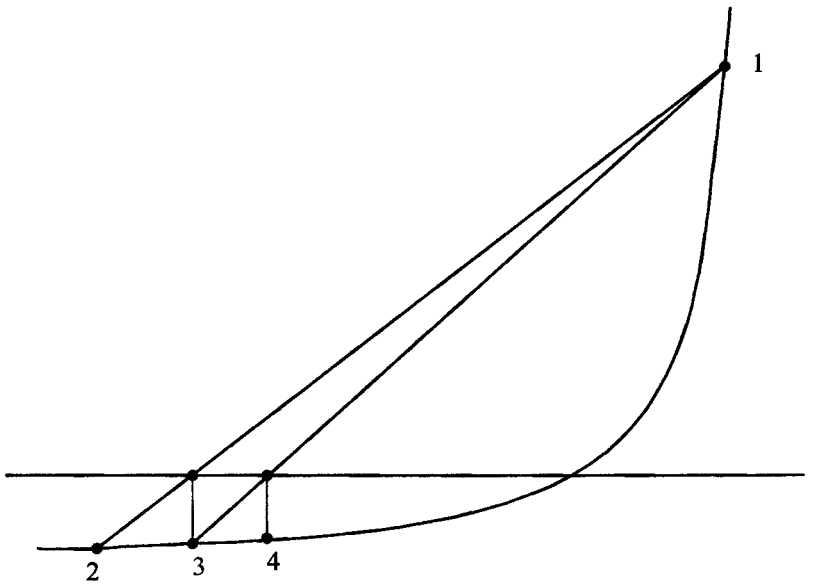
Точка (x_{n+1}, y_{n+1}) является следующим приближением к корню. При этом не заботятся о локализации корня, т. е. о знаках y_{n-1}, y_n, y_{n+1}. Следующую прямую строят по точкам (x_n, y_n) и (x_{n+1}, y_{n+1}). Если корень найден с погрешностью ε, то линейная интерполяция устраняет члены $O(\varepsilon)$, новая погрешность будет порядка $O(\varepsilon^2)$, т. е. сходимость к корню должна быть квадратичной (число верных знаков на каждом обороте удваивается). К сожалению, этот метод крайне ненадежен. В некоторых случаях он действительно дает квадратичную сходимость, но он может сходиться крайне медленно (медленнее деления пополам), а может и совсем не сойтись к корню:



Так что этот метод лучше не применять.

6.3. Линейная интерполяция с проверкой знаков

В отличие от метода линейной интерполяции без проверки знаков, в этом методе мы заботимся о локализации корня, т. е. y_{n-1} и y_n имеют разные знаки, поэтому x_{n+1} заведомо лежит между x_{n-1} и x_n, а в качестве следующей пары точек мы берем либо x_{n-1} и x_{n+1}, либо x_{n+1} и x_n, в зависимости от того, на каком отрезке локализован корень. Ясно, что возможное использование точки x_{n-1} вместо x_n не ускоряет сходимость. Зато оно гарантирует локализацию корня и сходимость. Однако, как и в предыдущем методе, сходимость может оказаться квадратичной, а может оказаться и крайне медленной (медленнее деления пополам):



Так что этот метод тоже лучше не применять.

6.4. Обратная квадратичная интерполяция

Этот метод является не двухточечным, а трехточечным и предназначен как раз для борьбы с теми типами корней, для которых непригодна линейная интерполяция.

По трем точкам (a, y_a) (b, y_b) (c, y_c) строится «обратная» парабола (полином второго порядка, но не $y(x)$, а $x(y)$):

$$x = a\frac{(y-y_b)(y-y_c)}{(y_a-y_b)(y_a-y_c)} + b\frac{(y-y_a)(y-y_c)}{(y_b-y_a)(y_b-y_c)} + c\frac{(y-y_a)(y-y_b)}{(y_c-y_a)(y_c-y_a)}.$$

После этого ищется ее корень, т. е. x_0, соответствующий $y = 0$:

$$x_0 = a\frac{y_b y_c}{(y_a-y_b)(y_a-y_c)} + b\frac{y_a y_c}{(y_b-y_a)(y_b-y_c)} + c\frac{y_a y_b}{(y_c-y_a)(y_c-y_a)}.$$

Ясно, что получив новое приближение к корню x_0, мы можем по-разному выбрать те три новые точки, по которым будет строиться следующая парабола. Простейший вариант — просто добавить x_0 и выкинуть самое

старое приближение к корню, не заботясь о знаках функции. Другой вариант — брать те три точки, которые образуют самый короткий отрезок, локализующий корень.

К сожалению, этот метод опять-таки не гарантирует сходимости не хуже линейной, так что лучше его не применять.

6.5. Метод Ньютона

Этот метод вообще одноточечный. В точке (x_{n-1}, y_{n-1}), являющейся предыдущим приближением к корню, строят касательную к функции:

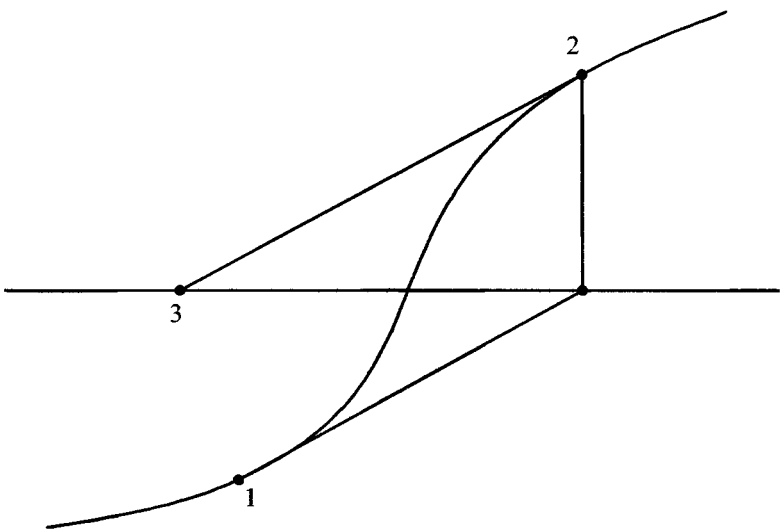
$$x = x_{n-1} + (y - y_{n-1})/y'_{n-1}$$

и находят ее корень:

$$x_n = x_{n-1} - y_{n-1}/y'_{n-1},$$

который и является следующим приближением к корню функции.

Теоретически сходимость метода должна быть квадратичной. Однако он может сходиться крайне медленно (сходимость медленнее линейной), а может и совсем не сойтись к корню:



Кроме того, здесь надо уметь вычислять не только саму функцию, но и ее производную (поскольку нередко приходится искать корни функций,

для которых невозможно выписать явное аналитическое выражение, то это может оказаться существенным недостатком метода).

Как ни странно, в большинстве систем аналитических вычислений встроенная функция численного поиска одномерных корней использует именно метод Ньютона. Опыт использования этих функций показывает, что этот метод действительно ненадежен.

Так что этот метод лучше не применять.

6.6. Адаптированный метод Брендта

Предлагаемый метод представляет собой упрощенный вариант метода Брендта. Метод не одношаговый, а двушаговый — из исходной пары точек, локализующей корень, за два шага строятся еще две точки, также локализующие корень. Идея состоит в том, что надо построить не одну точку, близкую к корню, а две точки, которые близки к корню, но находятся по разные стороны от него, что значительно ускоряет сходимость. Кроме того, в метод вставлены проверки, которые гарантируют сходимость не хуже линейной.

Алгоритм:

Первый шаг состоит в линейной интерполяции по двум исходным точкам a и b, соответствующий корень

$$c = [ay_b + by_a]/[y_a - y_b] \qquad y_c = f(x_c)$$

есть первая из двух новых точек.

Вторая из точек строится в зависимости от того, каков знак y_c. Если знаки y_c и y_a одинаковы,

то проводят линейную интерполяцию по точкам a и c, соответствующий корень

$$d = [ay_c + cy_a]/[y_a - y_c]$$

вычисляется и проходит следующие проверки:

Во-первых, d не должно быть больше $(c+b)/2$. Если $d > (c+b)/2$, то полагают $d = (c+b)/2$. Это гарантирует сходимость не хуже линейной — интервал, в котором локализован корень (а с учетом знака y_c это интервал $[c, b]$), убывает не менее чем в 2 раза.

Во-вторых, d не должно быть меньше, чем $c+\varepsilon$, где ε — заказанная точность. Если $d < c+\varepsilon$, то полагают $d = c+\varepsilon$. Дело в том, что метод рассчитан на то, что точки c и d лежат по разные стороны от корня и локализуют его (как правило это действительно так). Корень лежит правее c, т. е. брать точку d левее, чем $c+\varepsilon$, бессмысленно.

Наконец, вычисляется $y_d = f(x_d)$. В идеальном случае знаки y_d и y_c противоположны, т. е. корень локализован на отрезке $[c, d]$. Более того, точки c и d (они есть результат линейной интерполяции) лежат гораздо ближе к корню, чем исходные точки, так что «сходимость на данном шаге» в этом идеальном случае квадратична. В противном случае корень локализован на отрезке $[d, b]$. Это означает, что поведение функции на отрезке $[a, b]$ не может быть аппроксимировано линейной функцией с небольшой квадратичной добавкой. В этом случае «сходимость на данном шаге» не хуже линейной. Как правило, с уменьшением отрезка идеальный случай встречается все чаще, т. е. в среднем сходимость метода, как правило, квадратичная.

Если знаки y_c и y_a разные, т. е. знаки y_c и y_b одинаковые,

то делают все то же самое, с точностью до замены $a \leftrightarrow b$:

$$d = [by_c + cy_b]/[y_b - y_c].$$

Если $d < (c+a)/2$, то полагают $d = (c+a)/2$.

Если $d > c-\varepsilon$, то полагают $d = c-\varepsilon$.

Вычисляется $y_d = f(x_d)$. Если знаки y_c и y_d разные («идеальный случай»), то корень локализован на отрезке $[c, d]$, в противном случае он локализован на отрезке $[a, d]$.

После этого все повторяется, пока не будет достигнута заказанная точность.

Изложенный метод, как правило, обеспечивает квадратичную сходимость, но в любом случае сходимость не хуже линейной. Притом он в среднем быстрее большинства других квадратичных методов. Так что в том случае, когда скорость нахождения корня критична, его вполне можно применять.

Метод можно слегка улучшить, если ввести в него обратную квадратичную интерполяцию и добавить лишние проверки (после этого получится «настоящий» метод Брендта), однако, как показывает практика, ускорение оказывается не слишком значительным.

7. Многомерные корни

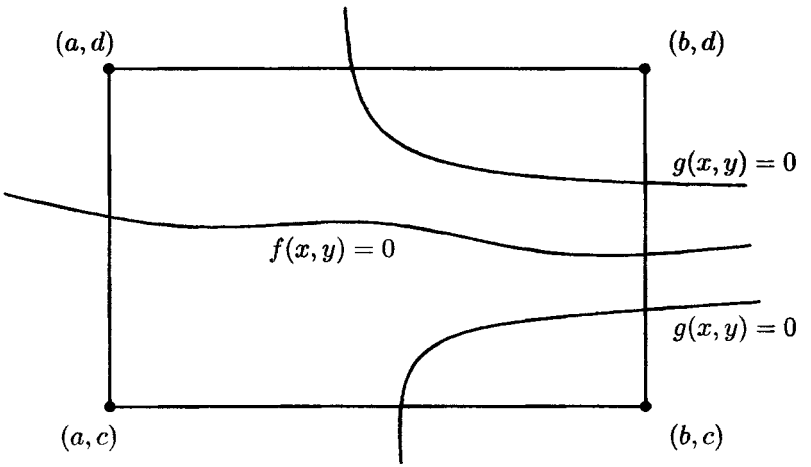
Задача поиска многомерного корня (решение системы N уравнений $f_i(\vec{x}) = 0$ с N неизвестными x_i) радикально отличается от поиска одномерного корня. Прежде всего, многомерный корень не локализуем. Рассмотрим, например, двумерный случай — систему $f(x,y) = 0$, $g(x,y) = 0$. Семейство решений первого уравнения представляет собой линию, разделяющую области положительных и отрицательных значений функции $f(x,y)$. Точно так же, семейство решений второго уравнения отделяет области знаков «+» и «−» для функции $g(x,y)$. Пересечение этих линий и есть искомый корень, т. е. корень является местом соприкосновения четырех секторов I, II, III и IV, в которых реализуются такие наборы знаков для функций $f(x,y)$ и $g(x,y)$:

IV $f<0,\ g>0$	III $f<0,\ g<0$
I $f>0,\ g>0$	II $f>0,\ g<0$

(разумеется, положение секторов указано с точностью до поворотов и зеркальных отражений, однако сектора I и III должны лежать «по диагонали» и соприкасаться в одной точке). Поэтому, казалось бы, корень можно локализовать четырьмя точками, лежащими в каждом из секторов. Например, пусть есть прямоугольник $(a,b)\times(c,d)$, вершины которого лежат в указанных секторах:

IV $f(a,d)<0,\ g(a,d)>0$	III $f(b,d)<0,\ g(b,d)<0$
I $f(a,c)>0,\ g(a,c)>0$	II $f(b,c)>0,\ g(b,c)<0$.

К сожалению, из этого совершенно не следует, что внутри прямоугольника есть корень:



Поэтому все те методы нахождения многомерных корней, которые будут изложены ниже, к сожалению, являются ненадежными — они могут сходиться к какому-нибудь корню, а могут и не сходиться. Если один корень уже найден, то единственный способ найти другой корень — это взять другое начальное приближение к корню. Что касается поиска начальных приближений к корням, то, разумеется, возможность провести табуляцию функций системы тем меньше, чем выше размерность пространства.

Метод Ньютона:

Это точный аналог одномерного метода Ньютона, т. е. одноточечный метод, в котором используется производная. В многомерном случае (система N уравнений для N неизвестных) необходимо уметь вычислять градиенты всех функций системы.

Начинаем с предыдущего приближения к корню, т. е. с точки

$$\vec{x}^{\,(n-1)} = (x_1^{(n-1)}, x_2^{(n-1)}, \ldots x_N^{(n-1)}).$$

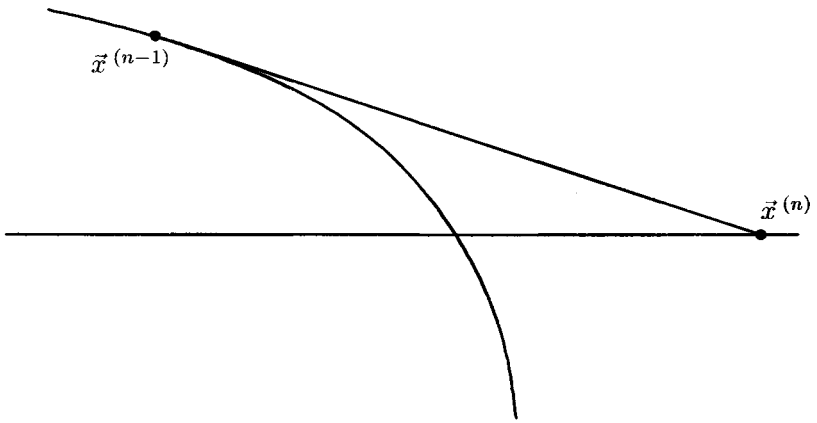
Тогда

$$f_j(\vec{x}^{\,(n-1)} + \delta\vec{x}) \approx f_j(\vec{x}) + \partial_i f_j(\vec{x}^{\,(n-1)})\delta x_i = 0.$$

Следовательно, новое приближение к корню $\vec{x}^{\,(n)} = \vec{x}^{\,(n-1)} + \delta\vec{x}$ выражается через смещение

$$\delta x_i = -\left[\partial_i f_j\left(\vec{x}^{\,(n-1)}\right)\right]^{(-1)} f_j\left(\vec{x}^{\,(n-1)}\right).$$

(Здесь необходимо решить систему линейных уравнений $N \times N$, см. раздел «Системы линейных уравнений».) При этом возникает такая же проблема, как и в одномерном методе Ньютона, — метод может вообще не сойтись к корню, даже когда этот корень есть, и мы находимся достаточно близко к нему. Это связано с тем, что вычисленное значение смещения оказывается слишком большим:



Следовательно, при реализации этого алгоритма следует установить некоторую максимальную длину Δ_{max} для вектора $\delta\vec{x}$ и при ее превышении укорачивать вектор: $\delta\vec{x} \to \Delta_{max}\delta\vec{x}/|\delta\vec{x}|$. Нет универсального способа выбора величины Δ_{max} — она зависит от конкретной задачи.

Несмотря на необходимость вычисления градиентов и решения системы линейных уравнений на каждом шаге, метод Ньютона остается одним из основных методов поиска многомерных корней.

Метод минимизации невязки:

Можно построить функцию невязки:

$$F(\vec{x}) = \sum_{i=1}^{N} f_i^2(\vec{x}),$$

которая не просто равна нулю на решении нашей системы, но еще и имеет там минимум. После этого мы начинаем искать не нуль, а минимум функции $F(\vec{x})$. Если в найденном минимуме $F(\vec{x})$ обращается в нуль — значит, мы нашли наш многомерный корень. На первый взгляд, никаких преимуществ такой подход не дает — надо искать многомерный минимум, плюс к этому мы можем найти совершенно бесполезный минимум, в котором невязка не равна нулю. Однако в действительности это не так.

Во-первых, существует широкий спектр надежных методов поиска минимума, которые обязательно и достаточно быстро сходятся к минимуму, если он есть (см. раздел «Многомерные минимумы»). Более того, некоторые из этих методов не требуют вычисления градиента минимизируемой функции. (Следует еще раз напомнить, что довольно часто мы не располагаем явным аналитическим выражением для $f_i(\vec{x})$.)

Во-вторых, такой подход является «неявной регуляризацией» для того случая, когда мы имеем дело с некорректно поставленной задачей. Подробнее это будет обсуждено позже, здесь мы ограничимся лишь коротким словесным комментарием:

Представим себе, что мы решаем обратную задачу электростатики: на плоскости в узлах квадратной сетки расположены заряды q (это изображает некоторое непрерывное распределение поверхностных зарядов), а мы задаем, скажем, вертикальную компоненту E_z напряженности электрического поля на некоторой высоте h над плоскостью. Допустим, что она известна в узлах такой же сетки. Наивный подход к задаче состоит в том, что мы пишем систему линейных уравнений, связывающую напряженность и заряды. Результат решения этой системы будет ужасен — при гладком поведении E_z на сетке поведение зарядов q на сетке гладким не будет, более того, абсолютная величина зарядов q будет совершенно непропорциональна величине поля. Вместо «правильного» поведения $0.9, 1.0, 1.1, \ldots$ мы получим $-10^5, +10^5, -10^5, \ldots$. Заметим, что такое поведение не связано с тем, что мы неверно решили систему линейных уравнений, — прямой подстановкой «неправильного» решения в систему легко убедиться, что невязка очень мала.

Физический смысл такого ответа вполне тривиален — череда близких друг к другу огромных зарядов противоположного знака создает почти нулевое и гладкое поле. Математический смысл также очевиден — у матрицы той системы линейных уравнений, которую мы решаем, есть (и не один) собственный вектор с почти нулевым (очень маленьким) собственным значением. Эти собственные вектора будут входить в ответ с очень большими коэффициентами, что и даст большой по величине и скачущий от узла к узлу ответ.

Так что не каждое гладкое распределение E_z в точности соответствует гладкому распределению q. Однако для каждого гладкого распределения E_z можно указать гладкое распределение q, которое дает «почти в точности» это распределение E_z. Канонический метод решения такой задачи будет изложен ниже. Он заключается в наложении дополнительного условия гладкости на решение.

Как ни странно, поиск решения этой задачи путем минимизации невязки (без какого-либо условия гладкости и без регуляризующего функционала) также даст гладкое решение, дающее почти в точности требуемое распределение E_z.

Так что метод минимизации невязки действительно включает в себя неявную регуляризацию задачи[4].

Метод итераций:

Нередко встречаются системы уравнений, которые имеют следующую структуру:

$$f_i(\vec{x}) = x_i - g_i(\vec{x}) = 0 \qquad \text{либо} \qquad f_i(\vec{x}) = A_{ij}x_j - g_i(\vec{x}) = 0$$

(второй случай, разумеется, сводится к первому, так что далее мы рассматриваем только первый случай). Формально говоря, любую систему можно записать в таком виде, но применение метода итераций целесообразно лишь тогда, когда это «естественная» структура задачи.

Располагая предыдущим приближением $\vec{x}^{\,(n-1)}$, можно построить следующее:

$$x_i^{(n)} = g_i(\vec{x}^{\,(n-1)}).$$

Если $\vec{x}^{\,(n)}$ совпадает или почти совпадает с $\vec{x}^{\,(n-1)}$, значит, мы нашли искомое решение. Разумеется, такая наивная процедура сходится лишь в ред-

[4]Надо еще заметить, что приведенный пример довольно нагляден, но не очень удачен, поскольку, ввиду трансляционной инвариантности функции Грина, решать эту задачу путем тривиального обращения линейной системы нецелесообразно.

чайших случаях. Фактически следует применять итерационную процедуру, «заторможенную» параметром ε:

$$x_i^{(n)} = x_i^{(n-1)}(1-\varepsilon) + \varepsilon g_i(\vec{x}^{\,(n-1)}).$$

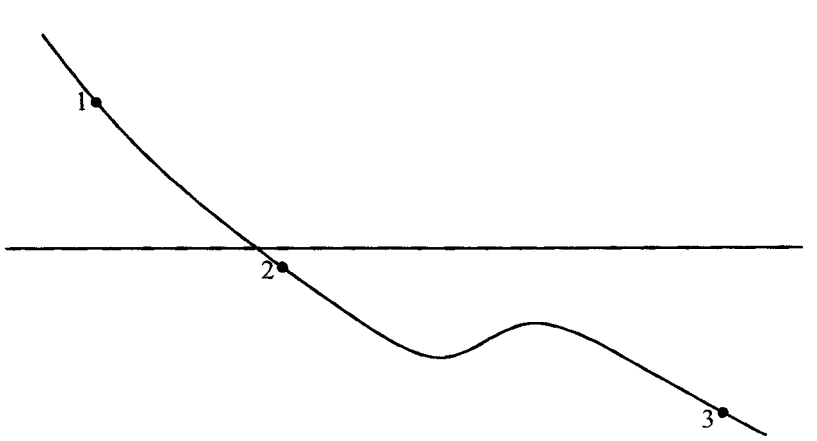
Здесь $\varepsilon < 1$ (например, 0.01) и подбирается в зависимости от задачи. Чем меньше ε, тем медленнее сходимость, но зато тем больше вероятность, что процедура сойдется.

В принципе, это самый быстрый и наименее хлопотный из методов (не надо вычислять градиенты), так что если он работает, то целесообразно применять именно его. Другое дело, что сходится он далеко не для всех задач.

8. Поиск одномерных минимумов

Одномерный минимум локализуется тремя точками: если точка b лежит между точками a и c, то условия $y_a > y_b$ и $y_c > y_b$ означают наличие минимума (или нескольких минимумов) на отрезке $[a, c]$.

Методы поиска одномерных минимумов (как и методы поиска одномерных корней) предполагают, что минимум уже локализован на некотором отрезке. Задача первоначальной локализации минимума решается столь же примитивно, как и задача первоначальной локализации корней. Мы начинаем с некоторой точки x_0, вычисляем в ней значение функции y_0, сдвигаемся (например) вправо на некоторое d (величина d зависит от конкретной задачи): $x_1 = x_0 + d$, вычисляем y_1 и, тем самым, определяем направление убывания функции. Допустим, она убывает вправо. Тогда осталось найти место, где убывание сменится возрастанием. Для этого достаточно вычислять функцию во все более удаленных вправо точках. Целесообразно (см. ниже) выбирать эти точки так, чтобы $(x_{n+1} - x_n)/(x_n - x_{n-1}) = (1-r)/r$, где $r = (3-\sqrt{5})/2$ — золотое сечение. Если очередное y_{n+1} окажется больше y_n, то минимум локализован точками x_{n-1}, x_n, x_{n+1}. Этот способ позволяет достаточно быстро локализовать минимум (если он есть), поскольку мы каждый раз сдвигаемся вправо на все большую величину (сдвиг растет экспоненциально, $(1-r)/r > 1$). С другой стороны, этот способ никак не гарантирует, что мы не пропустим некоторый локальный минимум:



Поэтому если необходимо найти все минимумы на некотором отрезке, то придется просто протабулировать функцию на этом отрезке с достаточно мелким шагом.

Следует также заметить, что точность, с которой можно найти минимум функции, пользуясь только значениями функции, гораздо меньше, чем точность, с которой можно найти корень. Количество верных знаков, которые можно получить при поиске корня, — это (почти) количество знаков, которое обеспечивает тип «double». Количество верных знаков при поиске минимума, как правило, приблизительно вдвое меньше. Дело в том, что функция $f(x)$ в окрестности минимума x_0 равна $f(x) \simeq f(x_0) + 0 + f''(x_0)(x - x_0)^2/2 + \ldots$, так что разницу в значениях функции можно заметить лишь при смещениях $x - x_0$, при которых заметна величина $(x - x_0)^2$. Если требуется более высокая точность при нахождении минимума, то следует искать корни производной.

8.1. Метод золотого сечения

Это аналог метода деления пополам.

Алгоритм:

Пусть три начальные точки расположены так, что $b = a + r * (c - a)$, где r — золотое сечение $r = (3 - \sqrt{5})/2$. При этом мы не предполагаем, что $a < c$ — может быть и наоборот. Однако в любом случае точка b лежит ближе к a, чем к c.

Строим новую точку $d = c + r * (a - c)$ и вычисляем $y_d = f(d)$.
Надо заметить, что при вычислении положения точки d возникает соблазн отказаться от умножения на r и вычислять d просто как $c - (b - a)$. Это совершенно неверная идея, т. к. экономия операций иллюзорна — одно умножение теряется на фоне вычисления функции f, — а потери могут оказаться фатальными. Дело в том, что величина отрезка, на котором локализован минимум, неуклонно уменьшается. Если мы получаем новую точку d умножением на r, то расстояние $|a-d|$ определено с той же точностью, что и $|a - c|$. Если мы получим d вычитанием, то $|a - d|$ является очень маленьким результатом многократного вычитания очень больших (по сравнению с $|a - d|$) чисел, т. е. содержит главным образом ошибки округления, а не правильное значение. В результате, если заказана достаточно высокая точность, то d может оказаться (и оказывается) вообще вне отрезка $[a, c]$.

Если $y_d < y_b$,

то минимум локализован на отрезке $[b, c]$ точками b, d, c ($y_d < y_c$, поскольку $y_b < y_c$). Следовательно, объявляем точку b новой точкой a: ($a = b$, $y_a = y_b$), а точку d новой точкой b: ($b = d$, $y_b = y_d$). Соглашение о том, что b лежит ближе к a, чем к c, при этом выполняется.

Если $y_d > y_b$,

то минимум локализован на отрезке $[a, d]$ точками a, b, d. Нам надо выполнить соглашение о том, что b лежит ближе к a, чем к c. Поэтому объявляем точку a новой точкой c: ($c = a$, $y_c = y_a$), а точку d новой точкой a: ($a = d$, $y_a = y_d$).

После этого все повторяется, пока не будет достигнута заказанная точность.

Единственное, что осталось проверить — что новые три точки делят новый отрезок в той же пропорции, как старые три точки делили старый отрезок. Иными словами,

$$|b - a|/|c - b| = |d - b|/|b - a| = |d - b|/|c - d|,$$

откуда

$$r/(1 - r) = (1 - 2r)/r = (1 - 2r)/r \quad \Rightarrow \quad r^2 - 3r + 1 = 0,$$

т. е.

$$r = (3 - \sqrt{5})/2 \qquad 1 - r = (-1 + \sqrt{5})/2.$$

Следовательно, отрезок действительно должен быть изначально разделен в пропорции золотого сечения. На каждом обороте отрезок укорачивается в $1 - r \approx 0.618 = 1/1.62$ раз. Сходимость метода линейная, он не сложнее метода деления пополам, так что если скорость поиска минимума не критична, то его вполне можно применять.

8.2. Адаптированный метод Брендта

Это четырехточечный и двушаговый метод. На каждом обороте находят две новые точки, которые по возможности лежат близко к минимуму и (вместе с предыдущими точками) локализуют его. При этом сходимость в идеальном случае квадратичная, но в любом случае не хуже, чем линейная.

Алгоритм:

Начать можно с 4-х точек золотого сечения. Итак, у нас есть упорядоченные точки a, b, c, d и значения функции в этих точках y_a, y_b, y_c, y_d, причем $y_a > y_b$ и $y_c < y_d$. Далее строим параболу, проходящую через три точки a, b, d, и находим ее минимум e. Аналогично, строим параболу по точкам a, c, d и ищем ее минимум f.

Парабола, проходящая через точки x_1, x_2 и x_3, определяется соотношением

$$y = y_1\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + \\ + y_3\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_1)}.$$

Ее производная равна

$$y' = y_1\frac{(2x-x_2-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2\frac{(2x-x_1-x_3)}{(x_2-x_1)(x_2-x_3)} + \\ + y_3\frac{(2x-x_1-x_2)}{(x_3-x_1)(x_3-x_1)}.$$

Следовательно, минимум достигается в точке

$$x_0 = \frac{1}{2}\,\frac{x_1^2(y_2-y_3) + x_2^2(y_3-y_1) + x_3^2(y_1-y_2)}{x_1(y_2-y_3) + x_2(y_3-y_1) + x_3(y_1-y_2)}.$$

После того как точки e и f найдены, их проверяют на приемлемость.

На положение точки e накладываются условия: $e > (a+b)/2$ и $e < (3c+d)/4$. При нарушении одного из условий точка e помещается на границу соответствующей разрешенной зоны.

На положение точки f накладываются условия: $f > (a+3b)/4$ и $f < (c+d)/2$. Опять-таки, при нарушении условия точка f помещается на соответствующую границу.

Наконец, проверяется условие $|f-e| > \epsilon$, где ϵ — заказанная точность нахождения минимума. При нарушении условия полагаем $f = e + \epsilon$ или $e = f + \epsilon$.

После этого упорядочиваем точки a, b, c, d, e, f в порядке возрастания и выбираем такие четыре новые точки a, b, c, d, что $y_a > y_b$ и $y_d > y_c$.

После этого все повторяется, пока не будет достигнута заказанная точность.

В идеальном случае каждые две новые точки e и f лежат гораздо ближе к минимуму, чем предыдущие две точки b и c, и минимум локализуется четверкой точек b, e, f, c или b, f, e, c. Так что сходимость оказывается квадратичной. Однако, благодаря проверкам, сходимость в любом случае будет не хуже линейной.

Если скорость нахождения одномерного минимума критична, то имеет смысл применять этот метод.

9. Многомерные минимумы

Методы поиска многомерных минимумов делятся на безградиентные и градиентные.

Следует сразу же подчеркнуть, что если нет способа вычислить градиент минимизируемой функции $F(\vec{x})$ (точно или хотя бы приблизительно), то *не следует* применять градиентный метод, вычисляя градиент численно покоординатным варьированием

$$\partial_i F(\vec{x}) = (F(x_1, x_2, \dots x_i + h, \dots, x_n) - F(x_1, x_2, \dots x_i, \dots, x_n))/h + O(h)$$

или

$$\partial_i F(\vec{x}) = (F(x_1, \dots x_i + h, \dots, x_n) - F(x_1, \dots x_i - h, \dots, x_n))/(2h) + O(h^2).$$

Лучше применить один из безградиентных методов.

Далее, не следует и мечтать о том, что хоть в каком-нибудь простейшем случае окажется применимым метод покоординатного спуска.

Наконец, не следует думать, что метод градиентного спуска является разумным методом поиска многомерного минимума.

Многие методы поиска многомерного минимума опираются на методы поиска одномерного минимума. Именно, если мы находимся в некоторой точке $\vec{x}$ и так или иначе выбираем некоторое направление в пространстве $\vec{s}$, то мы можем найти одномерный минимум вдоль этого направления, т. е. можем найти минимум функции одной переменной: $\tilde{F}(\lambda) = F(\vec{x} + \lambda\vec{s})$. Найденный таким образом одномерный минимум, как правило, является очередным приближением к искомому многомерному минимуму.

9.1. Метод амебы (безградиентный)

Алгоритм:

В N-мерном пространстве строится некоторая фигура с $N + 1$ вершинами.

Далее повторяют следующие действия:

Из всех вершин $\vec{x}_1, \ldots, \vec{x}_{N+1}$ выбирается вершина $\vec{h}$, в которой достигается максимальное значение функции $F(x)$ среди всех вершин; вершина $\vec{h}'$, в которой достигается максимальное значение функции $F(\vec{x})$ среди всех вершин за исключением $\vec{h}$, и вершина $\vec{l}$, в которой достигается минимальное значение функции $F(\vec{x})$ среди всех вершин.

Вычисляется новая точка $\vec{n}$ — точка, лежащая на продолжении медианы, проведенной из вершины $\vec{h}$, и расположенная на таком же расстоянии от середины противолежащей стороны (основания медианы) $\vec{m}$:

$$\vec{m} = \sum_{\vec{x}_i \neq \vec{h}} \vec{x}_i / N \qquad \vec{n} = \vec{m} + (\vec{m} - \vec{h}).$$

Если $F(\vec{n}) < F(\vec{h}')$,

то строится $\vec{n}' = \vec{m} + 2(\vec{m} - \vec{h})$. При этом если $F(\vec{n}') < F(\vec{h}')$,

то точка $\vec{h}$ заменяется на $\vec{n}'$ (объем фигуры увеличивается вдвое),

в противном случае

$\vec{h}$ заменяется на $\vec{n}$ (объем фигуры не меняется).

Если $F(\vec{n}) > F(\vec{h}')$,

то строится $\vec{n}'' = \vec{m} - (\vec{m} - \vec{h})/2$.

При этом если $F(\vec{n}'') < F(\vec{h}')$,

то точка $\vec{h}$ заменяется на $\vec{n}''$ (объем фигуры уменьшается вдвое).

В противном случае

заменяются на новые все точки, кроме $\vec{l}$: $\vec{x}_i \to \vec{l} + (\vec{x}_i - \vec{l})/2$, т.е. фигура сжимается вдвое в направлении вершины $\vec{l}$ (объем фигуры уменьшается вдвое). В этом случае функцию придется вычислять не в одной новой точке, а сразу в N точках.

Критерием сходимости является уменьшение объема фигуры при отсутствии движения в какую-либо сторону.

Поведение фигуры действительно напоминает движение амебы: на склоне, где есть определенный уклон в некотором направлении, она катится вниз, увеличиваясь в размерах (и, следовательно, делая все более крупные шаги в направлении убывания функции). В узком овраге она сжимается, чтобы не слишком налезать на крутые стенки, и катится по дну оврага в сторону убывания не очень быстро (это обычная проблема при поиске многомерных минимумов). Наконец, достигнув минимума, она начинает вокруг него сжиматься.

Очень простой и довольно неэффективный метод. Пожалуй, его применять не стоит.

9.2. Метод Пауэлла (безградиентный)

Алгоритм:

Строится набор векторов $\vec{s}_1, \ldots \vec{s}_n$. Он инициализируется базисными векторами $\vec{s}_1 = \vec{e}_1, \ldots \vec{s}_n = \vec{e}_n$. Выбирается некоторая начальная точка $\vec{p}_0$ — начальное приближение к минимуму.

Каждый шаг итерационной процедуры состоит из следующих действий:

Начиная с точки $\vec{p}_0$ в направлении $\vec{s}_1$, ищем одномерный минимум функции $\tilde{F}(\lambda) = F(\vec{p}_0 + \lambda\vec{s}_1)$, получаем точку $\vec{p}_1$.

Начиная с точки $\vec{p}_1$ в направлении $\vec{s}_2$, ищем одномерный минимум функции $\tilde{F}(\lambda) = F(\vec{p}_1 + \lambda\vec{s}_2)$, получаем точку $\vec{p}_2$.

...

Начиная с точки $\vec{p}_{n-1}$ в направлении $\vec{s}_n$, ищем одномерный минимум функции $\tilde{F}(\lambda) = F(\vec{p}_{n-1} + \lambda\vec{s}_n)$, получаем точку $\vec{p}_n$.

Начиная с точки $\vec{p}_n$ в направлении $\vec{s}^{\,*} = \vec{p}_n - \vec{p}_0$, ищем одномерный минимум функции $\tilde{F}(\lambda) = F(\vec{p}_n + \lambda\vec{s}^{\,*})$, получаем точку $\vec{p}_0^{\,*}$, которая будет начальной точкой $\vec{p}_0$ для следующей итерации.

Переприсваиваем набор векторов: $\vec{s}_1 = \vec{s}_2$, $\vec{s}_2 = \vec{s}_3$, ... $\vec{s}_{n-1} = \vec{s}_n$, $\vec{s}_n = \vec{s}^{\,*}$.

Через (приблизительно) n итераций набор векторов $\vec{s}_n$ вырождается — они становятся линейно зависимыми. Поэтому через каждые n (или $3n$) итераций набор векторов $\vec{s}_1, \ldots \vec{s}_n$ следует вновь инициализировать базисными векторами $\vec{s}_1 = \vec{e}_1, \ldots \vec{s}_n = \vec{e}_n$.

В этом методе происходит что-то вроде неявного вычисления градиента, так что он заметно эффективнее метода амебы. Поэтому при абсолютной невозможности найти (хотя бы приблизительно) градиент минимизируемой функции следует использовать именно метод Пауэлла.

9.3. Метод сопряженных градиентов (градиентный)

Прежде всего заметим, что метод, который, на первый взгляд, кажется наиболее естественным и эффективным — метод градиентного спуска, — применять *не следует*. Казалось бы, если мы фиксируем направление $\vec{s} = -\vec{\nabla}F(\vec{x})$, т. е. направление максимального убывания в данной точке $\vec{x}$, и находим новую точку как одномерный минимум вдоль этого направления, вновь ищем градиент в этой новой точке и т. д., то мы движемся в направлении минимума максимально быстро. К сожалению, это не так. Опыт непосредственных вычислений показывает, что если мы ищем минимум функции, профиль которой является уз-

ким оврагом с крутыми стенками, то такая процедура никогда не будет двигаться вдоль дна оврага в направлении убывания. Напротив, она будет прыгать с одной стенки на другую, причем каждое следующее движение будет почти противоположно предыдущему, и лишь в среднем (и очень медленно) мы будем смещаться в нужном направлении. Это связано с тем, что минимум вдоль градиента никогда не приходится в точности на дно оврага — он чуть залезает на противоположную стенку. Стенка крутая, поэтому «правильное» направление оптимального движения вдоль дна оврага маскируется большой поперечной составляющей. В результате на следующем шаге мы будем двигаться в основном поперек дна оврага (и только чуть-чуть вдоль дна). Поэтому мы снова слегка залезем теперь уже на противоположную стенку и т. д. Чем больше размерность пространства (количество аргументов функции), тем сильнее выражен этот эффект и тем медленнее будет сходимость.

Поэтому применять следует метод сопряженных градиентов, в котором этот недостаток хотя бы частично устранен. Одномерный минимум ищется не вдоль направления, противоположного градиенту, а вдоль такого направления, смещение вдоль которого не вызовет (почти не вызовет) необходимости смещаться обратно на следующем шаге.

Итак, пусть в процессе поиска минимума мы последовательно попадаем в точки $\vec{x}_{n-1}$, $\vec{x}_n$, $\vec{x}_{n+1}$. Введем обозначение $\vec{g}_k = -\vec{\nabla} f(\vec{x}_k)$. Сдвиги от точки к точке (т. е. направления, вдоль которых мы ищем одномерный минимум) есть $\vec{s}_n = \vec{x}_{n+1} - \vec{x}_n$, $\vec{s}_{n-1} = \vec{x}_n - \vec{x}_{n-1}$.

Поскольку вдоль $\vec{s}_n$ ищется одномерный минимум, который достигается в точке $\vec{x}_{n+1}$, то $\vec{s}_n \cdot \vec{g}_{n+1} = 0$; аналогично, $\vec{s}_{n-1} \cdot \vec{g}_n = 0$.

Мы ищем очередное направление $\vec{s}_n$ в виде $\vec{g}_n + \lambda_n \vec{s}_{n-1}$. При этом мы подбираем величину λ_n так, чтобы $\vec{g}_{n+1} \cdot \vec{s}_{n-1} = 0$, т. е. чтобы направление градиента в очередной точке $\vec{x}_{n+1}$ не вызывало необходимости каких-либо движений в направлении $\vec{s}_{n-1}$ (вдоль которого мы уже смещались и искали минимум). Для функции с профилем типа узкого оврага это должно предотвратить метание процедуры от стенки к стенке.

Трудность заключается в том, что мы не можем предсказать $\vec{g}_{n+1}$. Разложим нашу функцию в ряд с точностью до квадратичных членов:

$$F(\vec{x} + \delta\vec{x}) = F(\vec{x}) + \partial_i F(\vec{x}) \delta x_i + \frac{1}{2} \partial_i \partial_j F(\vec{x}) \delta x_i \delta x_j + \cdots$$

Введем обозначение для матрицы вторых производных, взятой с обратным знаком: $H_{ij}(x) = -\partial_i\partial_j f(\vec{x})$ и допустим, что она не очень сильно меняется от точки к точке, т. е. $H_{ij}(\vec{x}) \approx H_{ij}$. Тогда

$$\vec{g}(\vec{x}+\delta\vec{x}) = \vec{g}(\vec{x}) + \hat{H}\delta\vec{x}.$$

Следовательно,

$$\vec{g}_{n+1} - \vec{g}_n = \hat{H}\vec{s}_n \qquad \text{и} \qquad \vec{g}_n - \vec{g}_{n-1} = \hat{H}\vec{s}_{n-1}.$$

Тогда

$$0 = \vec{s}_{n-1}\cdot\vec{g}_{n+1} = \vec{s}_{n-1}\cdot(\vec{g}_n + \hat{H}\vec{s}_n) = \vec{s}_{n-1}\hat{H}\vec{s}_n.$$

Матрица $\hat{H}$ симметрична, так что

$$\vec{s}_{n-1}\hat{H}\vec{s}_n = \vec{s}_n\hat{H}\vec{s}_{n-1} = (\vec{g}_n + \lambda_n\vec{s}_{n-1})\cdot(\vec{g}_n - \vec{g}_{n-1}) =$$

$$= \vec{g}_n\cdot(\vec{g}_n - \vec{g}_{n-1}) + \lambda_n\vec{s}_{n-1}\cdot(\vec{g}_n - \vec{g}_{n-1}) = 0.$$

При этом

$$\vec{s}_{n-1}\cdot(\vec{g}_n - \vec{g}_{n-1}) = -\vec{s}_{n-1}\vec{g}_{n-1} = -(\vec{g}_{n-1} + \lambda_{n-1}\vec{s}_{n-2})\cdot\vec{g}_{n-1} = -\vec{g}_{n-1}\cdot\vec{g}_{n-1}.$$

Так что

$$\vec{g}_n\cdot(\vec{g}_n - \vec{g}_{n-1}) - \lambda_n\vec{g}_{n-1}\cdot\vec{g}_{n-1} = 0.$$

Окончательно получаем:

$$\lambda_n = \frac{\vec{g}_n\cdot(\vec{g}_n - \vec{g}_{n-1})}{\vec{g}_{n-1}\cdot\vec{g}_{n-1}}.$$

Итак, метод состоит попросту в том, что из очередной точки $\vec{x}_n$ мы ищем одномерный минимум в направлении $\vec{s}_n = \vec{g}_n + \lambda_n\vec{s}_{n-1}$ (здесь $\vec{g}_n = -\vec{\nabla}f(\vec{x}_n)$) и получаем следующее приближение к минимуму $\vec{x}_{n+1}$.

Это очень простой и довольно надежный метод, без подгоночных параметров. Однако для случая очень узких многомерных оврагов он все же не позволяет полностью устранить хаотичные метания от стенки к стенке. Кроме того, как всякий метод, опирающийся на значение функции, этот метод не позволяет найти минимум с высокой точностью (см. замечание про точность поиска одномерного минимума).

В конкретных задачах этот недостаток точности может оказаться фатальным. Рассмотрим типичную задачу на многомерный минимум, часто возникающую в

теоретической физике. Пусть мы ищем минимум функционала энергии некоторого поля:

$$H[\phi(x)] = \int dx\, [\frac{1}{2}\phi'(x) + V(\phi(x))].$$

Применяем решеточную аппроксимацию: $\phi_i = \phi(i * \Delta x)$ и получаем функцию

$$H(\vec{\phi}) = \sum_i \frac{1}{2}(\phi_{i+1} - \phi_i)^2/\Delta x + V(\phi_i)\Delta x,$$

минимум которой надо найти. При этом предполагается, что ϕ_i изображает непрерывную функцию, т. е. зависит от индекса i достаточно плавно, без скачков.

Чтобы уточнить профиль минимизируемой функции, целесообразно применить дискретное преобразование Фурье:

$$\phi_i \to \phi_k = \sum_n \phi_n \exp(2\pi i n k/N)/\sqrt{N}.$$

Оно диагонализует кинетический член:

$$\sum_i (\phi_{i+1} - \phi_i)^2/\Delta x \to \sum_k |\phi_k|^2 2(\cos(2\pi k/N) - 1)/\Delta x.$$

Отсюда видно, что профиль минимизируемой функции представляет собой именно узкий овраг с крутыми стенками — низкочастотные компоненты ($k \approx 0$ и $k \approx N$) имеют небольшую кинетическую энергию даже при заметной амплитуде ϕ_k, в то время как высокочастотные компоненты ($k \simeq N/2$ или $k \simeq N/3$, что соответствует ϕ_n вида $\phi_n \sim (-1)^n$ или $\phi_n \sim \cos(2\pi n/3)$) имеют очень большую кинетическую энергию даже при маленькой амплитуде ϕ_k.

Применение метода сопряженных градиентов позволит найти минимум с количеством верных знаков, приблизительно равным половине общего числа знаков, которые обеспечивает данная машина. Однако окончательный ответ будет содержать «высокочастотный мусор» с заметной амплитудой, а градиент в найденном минимуме будет заметно отличаться от нуля — даже незначительные смещения в направлении высокочастотных компонент (поперек дна оврага) приводят к значительному росту градиента.

Таким образом, мы получим не слишком гладкий и, следовательно, не слишком удовлетворительный ответ. Во всяком случае найти $\phi'(x)$, $\phi''(x)$ и уж тем более старшие производные поля с достаточной точностью будет невозможно.

Все это связано именно с тем, что мы руководствуемся значением функции, а не пытаемся занулить градиент.

9.4. Динамический метод (градиентный)

Будем думать о нашей функции как о потенциале в n-мерном пространстве, в котором движется материальная точка. Последняя движется с

трением. Градиент, взятый с обратным знаком, есть в точности сила. Ясно, что точка свалится в тот или иной локальный минимум потенциала. Следовательно, численно решая дифференциальное уравнение

$$\ddot{\vec{x}} = -\vec{\nabla}F(\vec{x}) - \gamma\dot{\vec{x}},$$

мы придем в некоторый локальным минимум.

Практически эта идея реализуется так:

Алгоритм:

выбираются два параметра — шаг по времени dt и коэффициент уменьшения скорости на каждом шаге $\beta < 1$. Например, $dt = 0.1$ и $\beta = 0.995$. Нам совершенно не нужно находить истинную траекторию движения к минимуму. Напротив, нам надо как можно скорее до него дойти, пусть и по не очень-то верной траектории. Поэтому dt надо выбирать максимально возможным, при котором процедура еще устойчива.

Выбираются начальная точка $\vec{x}$ и начальная скорость $\vec{v} = 0$. Сдвиг по времени на dt реализуется так:

$\vec{v} = \vec{v} - \vec{\nabla}F(\vec{x})dt$. (Это изменение скорости под действием силы. Вычисление силы — это вычисление градиента в данной точке.)

$\vec{v} = \beta\vec{v}$. (Это аналог трения. «Настоящее» трение должно было бы изображаться как $\vec{v} = \vec{v} - \gamma\vec{v}dt$, т. е. $\beta = 1 - \gamma dt$, где γ — «настоящий» коэффициент трения. Фактически, затухание определяется именно β, так что удобнее пользоваться двумя независимыми параметрами dt и β. В противном случае пришлось бы еще следить за тем, чтобы β было положительным, т. е. чтобы $dt\gamma$ было меньше единицы.)

$\vec{x} = \vec{x} + \vec{v}dt$. (Это сдвиг в пространстве на новую точку $\vec{x}$.)

Здесь существенна правильная последовательность шагов. Именно, смещение на новую точку $\vec{x} + \vec{v}dt$ реализуется не с использованием «старой» скорости, вычисленной на предыдущем обороте, а с использованием «новой» скорости. Анализ на устойчивость показывает, что процедура с правильной последовательностью шагов гораздо устойчивее. Практически это означает, что величину dt можно выбрать гораздо большей, чем при неправильной последовательности, т. е. двигаться к минимуму мы будем гораздо быстрее.

теоретической физике. Пусть мы ищем минимум функционала энергии некоторого поля:

$$H[\phi(x)] = \int dx\, [\frac{1}{2}\phi'(x) + V(\phi(x))].$$

Применяем решеточную аппроксимацию: $\phi_i = \phi(i * \Delta x)$ и получаем функцию

$$H(\vec{\phi}) = \sum_i \frac{1}{2}(\phi_{i+1} - \phi_i)^2/\Delta x + V(\phi_i)\Delta x,$$

минимум которой надо найти. При этом предполагается, что ϕ_i изображает непрерывную функцию, т. е. зависит от индекса i достаточно плавно, без скачков.

Чтобы уточнить профиль минимизируемой функции, целесообразно применить дискретное преобразование Фурье:

$$\phi_i \to \phi_k = \sum_n \phi_n \exp(2\pi i n k/N)/\sqrt{N}.$$

Оно диагонализует кинетический член:

$$\sum_i (\phi_{i+1} - \phi_i)^2/\Delta x \to \sum_k |\phi_k|^2 2(\cos(2\pi k/N) - 1)/\Delta x.$$

Отсюда видно, что профиль минимизируемой функции представляет собой именно узкий овраг с крутыми стенками — низкочастотные компоненты ($k \approx 0$ и $k \approx N$) имеют небольшую кинетическую энергию даже при заметной амплитуде ϕ_k, в то время как высокочастотные компоненты ($k \simeq N/2$ или $k \simeq N/3$, что соответствует ϕ_n вида $\phi_n \sim (-1)^n$ или $\phi_n \sim \cos(2\pi n/3)$) имеют очень большую кинетическую энергию даже при маленькой амплитуде ϕ_k.

Применение метода сопряженных градиентов позволит найти минимум с количеством верных знаков, приблизительно равным половине общего числа знаков, которые обеспечивает данная машина. Однако окончательный ответ будет содержать «высокочастотный мусор» с заметной амплитудой, а градиент в найденном минимуме будет заметно отличаться от нуля — даже незначительные смещения в направлении высокочастотных компонент (поперек дна оврага) приводят к значительному росту градиента.

Таким образом, мы получим не слишком гладкий и, следовательно, не слишком удовлетворительный ответ. Во всяком случае найти $\phi'(x)$, $\phi''(x)$ и уж тем более старшие производные поля с достаточной точностью будет невозможно.

Все это связано именно с тем, что мы руководствуемся значением функции, а не пытаемся занулить градиент.

9.4. Динамический метод (градиентный)

Будем думать о нашей функции как о потенциале в n-мерном пространстве, в котором движется материальная точка. Последняя движется с

сказано, такую точку можно найти гораздо точнее, чем точку, в которой значение функции минимально. Так что если вернуться к примеру из предыдущего раздела о поиске минимума энергии некоторого поля, то при использовании динамического метода мы заведомо получим более точный, без высокочастотного мусора (высокочастотные гармоники высветятся), и, следовательно, более гладкий ответ, который можно будет многократно дифференцировать. Притом мы получим его гораздо быстрее, чем другими методами.

Неправильные попытки улучшения динамического метода:

Надо быть крайне осторожным при попытках «улучшить» динамический метод. Идея метода настолько проста, что кажется, что возможности его улучшения практически безграничны. Перечислим основные соблазнительные идеи, большая часть которых позволяет существенно *ухудшить* сходимость.

Первая из идей заключается в том, что если вдруг по ходу итераций значение функции возросло (т. е. мы движемся «не туда»), то надо занулить скорость, а не ждать, пока скорость в этом «неправильном» направлении сама постепенно обратиться в нуль и поменяет знак на противоположный. Эта идея была бы правильной, если бы речь шла об одномерном минимуме. Уже в двумерном случае она очевидно неправильна — при повороте дна оврага «неправильна» поперечная (поперек дна) компонента скорости (из-за которой мы и залезли слегка на стенку), а накопленная продольная компонента «правильна». Лучше дать поперечной силе плавно повернуть скорость, чем занулить скорость совсем, т. к. после зануления точка будет еще долго разгоняться в «правильном» направлении до прежней скорости. Так что зануление скорости сильно замедляет сходимость.

Вторая из идей заключается в том, что надо по ходу дела менять dt (как в схемах решения обыкновенных дифференциальных уравнений с адаптивным изменением шага). Однако адаптивное изменение шага dt позволяет обеспечить точность вычисления траектории (в разных точках $\vec{x}$ требуется разное dt, чтобы получить одну и ту же точность), которая при поиске минимума нам совсем не нужна. Нам от dt требуется только устойчивость, а, как уже было сказано, максимально возможное dt, при котором еще нет разноса, почти не зависит от точки $\vec{x}$. Отклонение dt от максимально возможного в сторону увеличения означает разнос алгоритма (неограниченный рост градиента и убегание $\vec{x}$ на бесконечность) и необходимость возвращаться

назад. Отклонение dt от максимально возможного в сторону уменьшения означает замедление сходимости.

Третья из идей заключается в том, чтобы различать режим длительного движения в сторону минимума (когда оптимальным было бы значение β, близкое к единице, например, 0.995) и режим заключительных колебаний в окрестности минимума (когда оптимальным было бы значение β, гораздо меньшее единицы, например, 0.8). Режим колебаний детектируется или по существенному изменению скорости за несколько шагов, или по существенному изменению градиента за несколько шагов, или по разнице в направлениях у градиента и скорости. При выполнении одного из этих условий (т. е. в режиме колебаний) полагают $\beta = 0.8$, в противном случае полагают $\beta = 0.995$. К сожалению, на практике оказывается, что такая процедура (как правило) замедляет сходимость.

На самом деле третья идея почти правильна, неудачна только ее реализация. Дело в том, что динамический метод болезненно реагирует на все скачкообразные изменения своих параметров или принудительные изменения скорости или координаты. Если ввести плавное регулирование коэффициента трения, то сходимость можно заметно ускорить.

Улучшенный динамический метод:

Вычисляется косинус угла между «силой» и скоростью:

$$\cos(\theta) = -\frac{\vec{\nabla}F(\vec{x}) \cdot \vec{v}}{|\vec{\nabla}F(\vec{x})| \cdot |\vec{v}|},$$

после чего полагают коэффициент уменьшения скорости равным $\beta = B_0 * \cos(\theta)$, где $B_0 = 1.01 \cdots 1.02$, т. е. допускается даже некоторое ускорение. При этом если $\beta < B_1$, то полагают $\beta = B_1$, где $B_1 = 0.5 \cdots 0.7$. В противном случае β может оказаться слишком близкой к нулю (получится полная остановка) или вообще залезет в отрицательную область.

Динамический метод очень прост в написании. Единственным недостатком является необходимость подгонки параметров под данную задачу. Пожалуй, этот метод является оптимальным при поиске многомерных минимумов (по крайней мере для того класса задач, с которыми сталкивались авторы), так что именно его и следует применять.

10. Численное интегрирование

10.1. Разнообразные n-точечные формулы

Соорудить n-точечную интегральную формулу («квадратурную схему») можно следующим образом: берем n равномерно расположенных точек $x_1 \dots x_n$ и подбираем в них веса $c_1 \dots c_n$ так, чтобы для эталонных функций — полиномов $f_k(x) = x^k$ $(k = 0 \dots n-1)$ — выражения $\sum c_i f_k(x_i)$ совпадали с точными ответами для соответствующих интегралов $\int_{x_1}^{x_n} f_k(x)\,dx$. Для упрощения выкладок точки x_i целесообразно располагать симметрично относительно нуля, тогда уравнения для весов также окажутся симметричными (как и сами веса), в частности, уравнения для нечетных эталонных функций превратятся в тождества. Кстати, именно поэтому схемы с нечетным количеством точек n предпочтительнее — они дают правильный (нулевой) ответ для одной «лишней» эталонной функции $f_n(x)$ (нечетной).

1. Двухточечная схема: есть две точки $x_1 = -0.5h$, $x_2 = 0.5h$ (шаг равен h). Для $f_0(x) = 1$ имеем $c_1 + c_2 = h$; для $f_1(x) = x$ имеем $-0.5hc_1 + 0.5hc_2 = 0$. Так что, как и должно быть, веса симметричны, причем $c_1 = c_2 = 0.5h$. Погрешность схемы $O(h^2)$. Если состыковать много таких двухточечных отрезков, то получим формулу трапеций:

$$\int_a^b f(x) = I_h = h(0.5f_1 + f_2 + \dots + f_{n-1} + 0.5f_n).$$

   - Этой формулой можно было бы воспользоваться так: сосчитать эту сумму для некоторого h, затем уменьшить h вдвое $h \to h/2$ (именно вдвое, чтобы использовать те значения функции, которые уже сосчитаны), досчитать значения функции в недостающих точках (приходящихся на середины прежних отрезков) и сравнить I_h с $I_{h/2}$. Текущий ответ — это $I_{h/2}$, а разность $|I_h - I_{h/2}|$ позволяет оценить погрешность. Если точность недостаточна, следует еще раз поделить шаг пополам $h/2 \to h/4$ (и вновь досчитать только недостающие точки, приходящиеся на середины прежних отрезков длины $h/2$), и так далее. Поскольку есть столь же простой, но

гораздо более эффективный метод, пользоваться этим методом не стоит.

2. Трехточечная схема: есть три точки $x_1 = -h$, $x_2 = 0$, $x_3 = h$. Заведомо $c_1 = c_3$. Для $f_0(x) = 1$ имеем $c_1 + c_2 + c_3 = 2h$, для $f_1(x) = x$ имеем тождество, для $f_2(x) = x^2$ имеем $h^2c_1 + h^2c_3 = 2h^3/3$. Отсюда $c_1 = c_3 = h/3$, $c_2 = 4h/3$. Поскольку для $f_3(x) = x^3$ имеем тождество, то формула на один порядок точнее, чем должна была бы быть: ее погрешность будет не $O(h^3)$, а $O(h^4)$. Если состыковать много таких трехточечных отрезков, то получим метод Симпсона:

$$\int_a^b f(x) = I_h = (h/3)(f_1 + 4f_2 + 2f_3 + 4f_4 + \cdots + 4f_{n-1} + f_n).$$

- Многие полагают, что на этой формуле и следует остановиться. Это не так. Есть столь же простой, но гораздо более эффективный метод, так что пользоваться этим методом не стоит.

3. Четырехточечная схема: есть четыре точки $x_1 = -1.5h$, $x_2 = -0.5h$, $x_3 = 0.5h$, $x_4 = 1.5h$. Веса симметричны, т. е. $c_1 = c_4$, $c_2 = c_3$. Для $f_0(x)$ имеем $2c_1 + 2c_2 = 3h$, для $f_2(x)$ имеем $2c_1(1.5h)^2 + 2c_2(0.5h)^2 = 2(1.5h)^3/3$, т. е. $18c_1 + 2c_2 = 9h$. Отсюда $c_1 = c_4 = 3h/8$, $c_2 = c_3 = 9h/8$. Погрешность ничуть не лучше, чем в трехточечной схеме: $O(h^4)$.

4. Пятиточечная схема: есть пять точек $x_1 = -2h$, $x_2 = -h$, $x_3 = 0$, $x_4 = h$, $x_5 = 2h$. Веса симметричны, т. е. $c_1 = c_5$, $c_2 = c_4$. Для $f_0(x)$ имеем $2c_1 + 2c_2 + c_3 = 4h$, для $f_2(x)$ имеем $2c_1(2h)^2 + 2c_2(h)^2 = 2(2h)^3/3$, для $f_4(x)$ имеем $2c_1(2h)^4 + 2c_2(h)^4 = 2(2h)^5/5$. Отсюда $c_1 = c_5 = 14h/45$, $c_2 = c_4 = 64h/45$, $c_3 = 24h/45$. Погрешность опять же на один порядок лучше, чем должна была бы быть, — не $O(h^5)$, а $O(h^6)$.

Довольно ясно, что этот процесс можно продолжать до бесконечности. Надо понимать две вещи:

Во-первых, если при данном шаге формула Симпсона действительно, как правило, гораздо лучше формулы трапеций, то это не значит, что семиточечная схема настолько же лучше пятиточечной. Все равно при данном порядке схемы уменьшение шага сильнее влияет на точность, чем применение схемы более высокого порядка.

Во-вторых, не надо думать, что сила формулы Симпсона заложена в чередовании весов во внутренних точках интервала $(\ldots, 4/3, 2/3, 4/3, 2/3, \ldots)$. Легко соорудить схему с точно такой же погрешностью $O(h^4)$, но имеющую единичные веса во внутренних точках. Действительно, достаточно построить две интегральные формулы, из которых одна начинается на одном конце пятиточечной схемой, продолжается трехточечной схемой и завершается четырехточечной схемой — ее погрешность будет $O(h^4)$, а другая, наоборот, начинается четырехточечной схемой, продолжается трехточечной схемой и завершается пятиточечной схемой — ее погрешность также будет $O(h^4)$. Во внутренних точках обеих схем будет обычное Симпсоновское чередование весов, но там, где у одной интегральной формулы будет $4/3$, у другой будет $2/3$, и наоборот. Рассмотрим интегральную формулу, равную полусумме этих схем. Разумеется, ее погрешность по-прежнему будет $O(h^4)$, между тем во внутренних точках она будет иметь единичные веса: $(\ldots, 1, 1, 1, 1, \ldots)$. Все ее отличие от формулы трапеций заключается в достаточно прихотливом поведении весов на концах интервала интегрирования.

То же самое утверждение можно доказать для любой n-точечной схемы. Действительно, «пристегивая» схемы достаточно высоких порядков на концах интервала, мы для любой n-точечной схемы можем построить n интегральных формул точности $O(h^n)$, в которых веса во внутренних точках сдвинуты на $0, 1, \ldots, n-1$ позиций относительно друг друга. Следовательно, взяв среднее арифметическое этих формул, мы получим интегральную формулу той же точности $O(h^n)$, но с единичными весами во внутренних точках. От формулы трапеций она будет отличаться только поведением весов в окрестности концевых точек.

Использование любой из этих формул, включая формулу Симпсона, — неразумный способ вычисления интеграла. Следует помнить, что даже если Вы располагаете очень быстрой машиной, т. е. не ограничены по числу операций, то применение неэффективного алгоритма при очень мелком шаге вовсе не гарантирует точного ответа — мелкий шаг означает большое количество операций, т. е. большую накопленную ошибку округления.

10.2. Алгоритм Ромберга

Алгоритм Ромберга является интерполяционным алгоритмом, который использует простейшую из квадратурных схем — двухточечную. Фактиче-

ски реализуется тот метод, который уже был изложен для формулы трапеций, но с небольшим добавлением, которое и делает метод эффективным.

Итак, по формуле трапеций вычисляются интегральные суммы I_h, $I_{h/2}$, $I_{h/4}$, ... (Как уже было сказано, при вычислении каждой следующей интегральной суммы не надо второй раз считать значения функции в тех точках, которые входят в предыдущую сумму.)

На каждом очередном шаге мы вычисляем интегральную сумму, соответствующую данному шагу. Следовательно, у нас есть не просто текущая интегральная сумма, а зависимость этой суммы от шага. Естественно попытаться проинтерполировать эту зависимость в точку, соответствующую нулевому шагу, т. е. точному ответу. Именно в этом и заключается метод Ромберга.

Единственная существенная техническая деталь — в качестве аргумента x при интерполяции зависимости $y(x)$ следует брать не сам шаг h, а его квадрат. Это связано с тем, что разложение зависимости $I(h)$ в ряд по степеням h в окрестности нуля включает только четные степени h (погрешность двухточечной схемы $O(h^2)$, а следующее слагаемое — не $O(h^3)$ (нечетные эталонные функции $f_n(x)$ мы всегда интегрируем точно), а $O(h^4)$ и т. д.). Опыт показывает, что попытка интерполировать не по h^2, а по h действительно замедляет сходимость.

Итак, на каждом очередном k-м шаге мы по k точкам ($x_i{=}h_i^2$, $y_i{=}I(h_i)$), $i = 1, \ldots, k$ (здесь $h_i = h_0/2^i$) проводим полиномиальную интерполяцию функции $y(x)$ в точку $x = 0$ (в нулевой шаг) и получаем ответ I'_k. Именно этот результат интерполяции мы рассматриваем как текущий ответ для интеграла, а погрешностью мы считаем разницу между этим ответом и ответом, полученным на предыдущем шаге: $|I'_k - I'_{k-1}|$. Как только погрешность оказывается меньше заказанной точности, мы обрываем процедуру.

Сходимость метода поражает воображение. Она достигается при настолько малом количестве точек (и, соответственно, большом шаге h), что не только формула трапеций, но и формула Симпсона еще очень далека от сходимости. Если заказанная точность близка к машинному нулю, то метод Симпсона может не сойтись вообще (при очень маленьком шаге будет слишком много вычислений функции и накопится большая ошибка округления), в то время как метод Ромберга вполне может сойтись.

За счет чего достигается эффективность метода Ромберга? Одна из формальных причин такова: уже на первом шаге мы получаем интерполяцию по двум точкам ($x_1 = h^2$, $y_1 = I(h)$) и ($x_2 = h^2/4$, $y_2 = I(h/2)$):

$$\begin{aligned} I_2' &= (4/3)I(h/2) - (1/3)I(h) = \\ &= (4/3)(h/2)\Big[(1/2)f_1 + f_2 + f_3 + f_4 + f_5 + \ldots\Big] - \\ &\quad - (1/3)(h)\Big[(1/2)f_1 + f_3 + f_5 + \ldots\Big] = \\ &= (h/2)\Big[(1/3)f_1 + (4/3)f_2 + (2/3)f_3 + (4/3)f_4 + (2/3)f_5 + \ldots\Big]. \end{aligned}$$

Это в точности совпадает с формулой Симпсона, имеющей точность $O(h^4)$, а не $O(h^2)$. Довольно ясно, что на следующем шаге мы (неявно) получим формулу с точностью $O(h^6)$ и т. д. (Заметим, что высокий порядок точности достигается без применения соответствующих n-точечных схем.) В действительности дело не столько в повышении порядка точности (существует много разных формул высокого порядка), сколько именно в интерполяции в нулевой шаг.

Итак, при необходимости вычислить интеграл следует применять именно алгоритм Ромберга.

10.3. Возможности переменного шага

Разумеется, метод Ромберга надо применять осмысленно. Именно, если функция почти постоянна в одной области и резко меняется в другой, то лучше использовать метод Ромберга два раза — по отдельности для каждой из областей. (Плавную область он пройдет с большим шагом, а область резкого изменения — с маленьким.)

Можно ли это автоматизировать? В принципе, можно — ведь вычисление определенного интеграла $\int\limits_a^b f(x)dx$ можно рассматривать как решение дифференциального уравнения $y' = f(x)$ с начальным условием $y(a) = 0$. Искомый интеграл есть в точности $y(b) = \int\limits_a^b f(x)dx$. Разработаны методы численного решения дифференциальных уравнений, допускающие автоматическое изменение шага в зависимости от свойств функции. Почему бы не применять их при вычислении интегралов?

Дело в том, что при решении дифференциального уравнения контролируется только локальная точность интегрирования. Глобальная точность (т. е. точность искомого интеграла) есть локальная точность, умноженная на число шагов, которое заранее неизвестно. Так что для оценки погрешности искомого интеграла придется решать дифференциальное уравнение несколько раз, задавая различную локальную точность. Если при заметном (как минимум в 1.5 раза) уменьшении шага ответ почти не меняется, то можно считать, что интеграл вычислен правильно.

Опыт показывает, что разумнее все же использовать алгоритм Ромберга.

10.4. Метод Гаусса

Прежде всего, напомним, как можно построить семейство классических ортогональных полиномов (КОП). Как известно, семейство КОП определяется интервалом $[a, b]$ (обе границы могут убегать на бесконечность) и весом $\rho(x)$.

Определяя скалярное произведение двух функций $\psi(x)$ и $\chi(x)$ как $\langle\psi|\chi\rangle = \int_a^b \rho(x)\psi(x)\chi(x)\,dx$, полагая $P_{-1} = 0$, $P_0 = 1$, мы можем легко построить все остальные полиномы просто из условия ортогональности. Мы не будем нормировать полиномы, зато мы будем считать, что коэффициент при старшей степени x равен 1. Итак, строим формулу рекурсии:

$$P_{n+1} = xP_n + a_nP_n + b_nP_{n-1}.$$

Тогда, скалярно умножая это соотношение на P_n, получим

$$\langle P_n|P_{n+1}\rangle = 0 = \langle P_n|x|P_n\rangle + a_n\langle P_n|P_n\rangle + b_n\langle P_n|P_{n-1}\rangle,$$

откуда

$$a_n = -\langle P_n|x|P_n\rangle/\langle P_n|P_n\rangle.$$

Аналогично, умножая это соотношение на P_{n-1}, получим

$$\langle P_{n-1}|P_{n+1}\rangle = 0 = \langle P_{n-1}|x|P_n\rangle + a_n\langle P_{n-1}|P_n\rangle + b_n\langle P_{n-1}|P_{n-1}\rangle,$$

откуда

$$b_n = -\langle P_{n-1}|x|P_n\rangle/\langle P_{n-1}|P_{n-1}\rangle.$$

Числитель этого соотношения можно упростить

$$\langle P_{n-1}|x|P_n\rangle = \langle P_n|P_n\rangle - a_{n-1}\langle P_{n-1}|P_n\rangle - b_{n-1}\langle P_{n-2}|P_n\rangle,$$

откуда окончательно получаем

$$b_n = -\langle P_n|P_n\rangle/\langle P_{n-1}|P_{n-1}\rangle.$$

Полезно напомнить, что для КОП строятся формула Родрига (явное выражение через n-ю производную) и производящая функция (функция двух аргументов, степенной ряд по первому аргументу дает как раз нужные полиномы по второму аргументу). Кроме того, большинство «наиболее популярных» КОП являются частными случаями гипергеометрической функции (или вырожденной гипергеометрической функции), так что для них известны явные выражения для коэффициентов при степенях x.

Рецепт интегрирования по Гауссу состоит в следующем:

Строим $N+1$ штук полиномов (от P_0 до P_N), принадлежащих некоторому семейству КОП. Ищем все N корней $(x_1, \ldots x_N)$ полинома P_N. Далее, решаем[5] систему линейных уравнений $N \times N$ относительно весов a_i:

$$\sum_{i=1}^{N} \rho(x_i)P_0(x_i)a_i = \int_a^b \rho(x)\,dx$$

$$\sum_{i=1}^{N} \rho(x_i)P_k(x_i)a_i = 0, \qquad k = 1, \ldots N-1.$$

Располагая точками x_i и коэффициентами a_i, мы аппроксимируем произвольный определенный интеграл на отрезке $[a, b]$ суммой:

$$\int_a^b f(x)\,dx \simeq \sum_{i=1}^{N} a_i f(x_i).$$

Очевидно, что это выражение для интеграла даст точный ответ для $f(x)$, совпадающей с любой линейной комбинацией первых $N+1$ полиномов (от P_0 до P_N), умноженной на $\rho(x)$[6]. Так что любая функция $f(x)$, которая обладает тем свойством, что отношение $f(x)/\rho(x)$ «хорошо раскладывается» по данному семейству КОП (т. е. коэффициенты разложения

[5]Можно показать, что решением системы будет

$$a_i = \langle P_{N-1}|P_{N-1}\rangle/\left[P_{N-1}(x_i)P'_{N-1}(x_i)\right].$$

[6]В действительности можно доказать, что оно даст точный ответ для любой линейной комбинации первых $2N$ полиномов (от P_0 до P_{2N-1}), умноженной на $\rho(x)$.

достаточно быстро убывают с ростом номера полинома), методом Гаусса будет проинтегрирована правильно.

Разумеется, далеко не всегда отрезок, на котором вычисляется интеграл, совпадает с отрезком, на котором определены КОП. Эта проблема решается тривиальным линейным преобразованием переменной интегрирования x.

Заметим, что метод Гаусса естественным образом позволяет находить коэффициенты разложения произвольной функции по данному семейству КОП. Действительно, коэффициент разложения $c_k = \int_a^b dx\ y(x)P_k(x)\rho(x)$ определяется интегралом как раз «подходящего» типа. Это позволяет проводить интерполяцию с помощью произвольного семейства КОП (см. раздел «Интерполяция»).

Особую роль играют полиномы Лежандра, для которых отрезок конечен, а вес равен единице: $a = -1$, $b = 1$, $\rho(x) = 1$. Большинство библиотечных функций, реализующих Гауссово интегрирование, используют именно полиномы Лежандра.

Удобны также и полиномы Чебышева (см. раздел «Интерполяция»), для которых известны явные выражения как для корней $x_i = \cos(\pi(i-1/2)/N)$, так и для коэффициентов $a_i = (\pi/N)\sin(\pi(i-1/2)/N)$. Казалось бы, при использовании метода Гаусса для интегрирования по конечному интервалу только их и следует применять: не надо находить корни x_i и не надо решать систему линейных уравнений для a_i. К сожалению, это не совсем так. Дело в том, что полиномы Чебышева определены на конечном отрезке, но их вес имеет корневую особенность: $a = -1$, $b = 1$, $\rho(x) = 1/\sqrt{1-x^2}$. Поэтому они оптимальны в том случае, когда подынтегральная функция имеет вид $f(x) = g(x)/\sqrt{1-x^2}$, если функция $g(x)$ хорошо раскладывается по полиномам Чебышева, и не так уж хороши во всех остальных случаях.

Впрочем, для полиномов Чебышева нетрудно вычислить и альтернативные веса $\tilde{a}_i$, которые дадут точный ответ для интеграла от любой линейной комбинации первых N полиномов $T_0, \ldots, T_{N-1}$, без умножения на вес. Действительно, располагая коэффициентами c_l разложения функции $f(x)$ по полиномам Чебышева:

$$c_l = \sum_{k=0}^{n-1} f(x_k)T_l(x_k)/M_l,$$

где $M_l = (1+\delta_{l0})\ N/2$, мы можем найти неопределенный (и, следовательно, определенный) интеграл от нее (см. раздел «Интерполяция»):

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{l=0}^{N-1} d_l[T_l(1) - T_l(-1)],$$

где $d_0 = const$ и $d_l = (c_{l-1} - c_{l+1})/(2l)$ при $l \geqslant 1$. Следовательно,

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{m=0}^{(N-1)/2} [1/(2m+1) - 1/(2m-1)]c_{2m},$$

Откуда окончательно получаем

$$\tilde{a}_i = \sum_{m=0}^{(N-1)/2} [1/(2m+1) - 1/(2m-1)]T_{2m}(x_i)/M_{2m}.$$

Легко проверить, что веса $\tilde{a}_i$ довольно близки к «каноническим» для полиномов Чебышева весам $a_i = (\pi/N)\sin(\pi(i-1/2)/N)$, более того, стремятся к ним при $N \to \infty$. Тем не менее, при интегрировании полинома (константы, например) «канонические» веса a_i дадут ответ с некоторой погрешностью (порядка 10^{-3} при $N = 20$), в то время как альтернативные веса $\tilde{a}_i$ дадут точный ответ.

Так что в методе Гаусса при интегрировании по конечному интервалу иногда целесообразно использовать не полиномы Лежандра, а полиномы Чебышева с альтернативными весами.

При данном шаге (точнее, данном количестве точек N, в которых вычисляется подынтегральная функция) метод Гаусса, как правило, точнее других методов. Формальная причина очень проста — для N точек метод Гаусса представляет собой N-точечную схему, которая дает точный ответ для $2N$ эталонных функций — полиномов $P_0, \ldots, P_{2N-1}$, умноженных на вес.

Платой за это является необходимость вычислять положение корней x_i и соответствующие веса a_i. Кроме того, в этом методе довольно затруднителен контроль точности: чтобы изменить шаг (количество точек N), придется заново находить x_i и a_i. Впрочем, можно разбить отрезок интегрирования

пополам и на каждой из половин применить метод Гаусса с прежним количеством точек N (тем самым общее количество точек возрастет вдвое, но без дополнительных хлопот). Однако следует иметь в виду, что функция, которая хорошо раскладывается по данному семейству КОП на исходном отрезке, вообще говоря, не обязана столь же хорошо раскладываться на половинках этого отрезка.

В целом рекомендация такова: если необходимо просто вычислить определенный интеграл, то метод Гаусса лучше не применять. Дополнительные хлопоты не окупаются некоторым выигрышем в точности. Однако в тех случаях, когда необходимо бороться за минимальное количество вызовов интегрируемой функции, например, если эта функция вычисляется с исключительно большим трудом, или при решении интегральных уравнений (для интегральных уравнений число точек, в которых мы вычисляем подынтегральную функцию, есть число неизвестных в системе уравнений, которую потом придется решать), следует использовать именно метод Гаусса.

10.5. Несобственные интегралы

Основной рецепт при взятии несобственных интегралов очень прост: не следует брать их численно.

Дело в том, что любой сходящийся несобственный интеграл можно так или иначе превратить в регулярный.

<u>При интегрировании функций с особенностями:</u>
все определяется характером особенности:

- Ложная несобственность, например: $\int\limits_0^1 \sin(x)/x\ dx$ — следует просто доопределить функцию в точке $x = 0$.
- Интегрируемая особенность, например: $\int\limits_0^1 1/\sqrt{x}\ dx$ — замена переменной интегрирования $x = t^2$ превратит несобственный интеграл в регулярный интеграл $\int\limits_0^1 2dt$.

В литературе можно встретить рецепты, опирающиеся на n-точечные схемы, в которых не используется концевая точка (в которой у

интегрируемой функции особенность). Например, двухточечная схема:

$$\int_{x_0}^{x_0+h} f(x)\,dx \approx h\,f(x_0+h/2)$$

или трехточечная схема:

$$\int_{x_0}^{x_0+h} f(x)\,dx \approx h\Big[(3/2)f(x_0+h)-(1/2)f(x_0+2h)\Big]$$

и т. п. Однако опыт показывает, что лучше все эти рецепты не применять (слишком велик риск потери точности), а преобразовать искомый интеграл.

При бесконечных пределах интегрирования:

все зависит от поведения подынтегральной функции на бесконечности:

- Ложная несобственность, например: $\int_0^\infty \exp(-x^2)\,dx$ — интегрируемая функция убывает достаточно быстро; легко оценить верхний предел интегрирования, дальше которого залезать не имеет смысла. В этом случае интегрирование в действительности идет по конечному интервалу. Для страховки можно увеличить выбранный верхний предел интегрирования раза в полтора и убедиться, что изменение ответа не превышает заказанную точность. Если это не так, т. е. первоначальная оценка верхнего предела неправильна, то следует повторить процедуру, т. е. снова увеличить верхний предел раза в полтора, и так до тех пор, пока не будет достигнута заказанная точность.

- Медленно сходящийся на бесконечности интеграл, например: $\int_1^\infty 1/x^2\,dx$ — замена переменной интегрирования $x = 1/t$ превратит его в регулярный интеграл $\int_0^1 1\,dt$. Другой возможностью является вычитание главной асимптотики из подынтегральной функции. Как

правило, асимптотику можно проинтегрировать аналитически, а оставшаяся часть будет сходиться достаточно быстро, см. п. в.).

Довольно ясно, что в случае $\int_1^\infty 1/x^2\,dx$ попытка взять «большой» верхний предел интегрирования, а потом увеличивать его в полтора (или два) раза, пока ответ не перестанет меняться, обречена на провал, т. к. при данном верхнем пределе X остаточный член будет порядка $1/X$, и достигнуть хорошей точности за конечное время невозможно. Для интеграла типа $\int_1^\infty \sin(x)/x\,dx$ ситуация еще хуже.

Итак, в случае несобственных интегралов разумнее не мучить машину, а самому преобразовать задачу к регулярному виду.

10.6. Многомерные интегралы

В случае не очень большого количества измерений (приблизительно до 5) и не слишком витиеватой поверхности, ограничивающей область интегрирования, рецепт очень прост: многомерный интеграл должен браться как повторный. В качестве дополнительной проверки можно еще и поменять порядок интегрирования (сначала по x, затем по y, а потом наоборот — сначала по y, затем по x) и сравнить ответы. При интегрировании по каждому измерению следует применять алгоритм Ромберга. То есть алгоритм Ромберга применяется, во-первых, для внутреннего интеграла $g(x_k) = \int f(x_k, y)\,dy$ в каждой точке x_k, нужной для внешнего интеграла, во-вторых, для самого внешнего интеграла $\int g(x)\,dx$. Единственная техническая деталь — при взятии внутреннего интеграла следует заказывать точность выше, чем необходимая точность для внешнего. Так что при каждом переходе к более внутреннему интегралу следует увеличивать точность в несколько раз. Иногда в литературе рекомендуют увеличивать точность на порядок, но опыт показывает, что это слишком много, достаточно увеличивать ее раза в три (в некоторых случаях, например, для гладких функций при плоской границе, ее можно и вовсе не увеличивать).

Иногда возникает мысль соорудить многомерный аналог алгоритма Ромберга, т. е. ввести квадратную (кубическую) сетку с шагом h, вычислить интегральную сумму сразу для многомерного интеграла, а потом устроить такую же интерполяцию по h в нулевой шаг, как в алгоритме Ромберга.

Идея эта совершенно неправильная. Опыт показывает, что такой алгоритм, как правило, сходится гораздо медленнее, чем повторный интеграл. Одна из причин такого поведения очень проста: бывают функции, которые при некоторых значениях x почти не зависят от y (внутренний интеграл по y сойдется очень быстро), а при других значениях x существенно зависят от y (внутренний интеграл по y сойдется только при очень маленьком шаге). Тем самым при взятии повторного интеграла достигается некоторая экономия: в тех областях, где это возможно, шаг интегрирования берется побольше. При вычислении интегральной суммы для многомерного интеграла шаг везде одинаков, а сходимость (шаг) определяется «самой плохой» областью, так что нам придется везде (в том числе и там, где это необязательно) брать очень маленький шаг, т. е. никакой экономии не будет.

В случае очень большого количества измерений M или очень сложной границы (когда трудно выписать пределы интегрирования для повторных интегралов) изложенный рецепт не годится. Как правило, в этом случае рекомендуют использовать метод Монте-Карло. Метод заключается в следующем: в область $\mathcal{V}$, по которой идет интегрирование, набрасывают случайные точки $\vec{x}^{(k)}$. Практически точки набрасывают, разумеется, не в саму область, а во включающий ее прямоугольник ($a_i \leqslant x_i^{(k)} \leqslant b_i$, $i = 1, \ldots, M$). Каждая координата $x_i^{(k)}$ генерируется как случайное число, равномерно распределенное на отрезке $[a_i, b_i]$. Если точка попадает в область, ее принимают, в противном случае ее отбрасывают. Довольно ясно, что

$$\int_{\mathcal{V}} f(\vec{x})\, d\vec{x} \approx \frac{V}{N} \sum_{k=1}^{N} f(\vec{x}^{(k)})$$

(здесь V — объем области $\mathcal{V}$), причем относительная погрешность пропорциональна $1/\sqrt{N}$, что не слишком воодушевляет. Если объем фигуры неизвестен, то его можно найти с той же относительной погрешностью $1/\sqrt{N}$ — объем V есть объем включающего прямоугольника, умноженный на долю принятых точек.

Заметим, что метод Монте-Карло требует очень качественного генератора случайных чисел.

Метод Монте-Карло не способен обеспечить очень высокую точность, так что его следует применять только в тех случаях, когда ничего другого не остается.

11. Ряды, произведения, цепные дроби

Ситуация с бесконечными рядами и произведениями напоминает ситуацию с интегрированием на бесконечном интервале.

Прежде всего заметим, что нет никакой необходимости рассматривать бесконечные произведения отдельно:

$$\log \prod_{n=1}^{\infty} a_n = \sum_{n=1}^{\infty} \log a_n,$$

при этом a_n должны стремиться к единице при $n \to \infty$, так что $\log a_n \approx a_n - 1$. Поэтому все рецепты, относящиеся к рядам, можно применить и к бесконечным произведениям с соответствующей незначительной модификацией.

Если ряд сходится достаточно быстро, то можно взять достаточно большой индекс N в качестве верхнего предела, вычислить конечную сумму $S_N = \sum_{n=1}^{N} a_n$. Чтобы оценить погрешность этого ответа, следует увеличить N в c раз: $N \to c \cdot N$ (c порядка полутора-двух), и сравнить ответы. «Довесок» $\sum_{n=N+1}^{c \cdot N} a_n$, который только и надо при этом сосчитать, рассматривается как погрешность для текущего ответа $S_{c \cdot N}$. Если точность недостаточна, то следует вновь увеличить верхний предел суммирования в c раз, и так далее, пока не будет достигнута нужная точность. При этом, как правило, можно оценить остаточный член $\sum_{n=N+1}^{\infty} a_n \approx \int_{N}^{\infty} a(x)\, dx$ по асимптотике членов ряда. Если этот примитивный алгоритм обеспечивает удовлетворительную скорость, то нет нужды применять что-либо еще. К сожалению, обычно скорость оказывается недостаточной.

Если сходимость медленная, то можно попытаться вычесть из членов ряда их главную асимптотику. Ряд, соответствующий асимптотике, нередко можно просуммировать аналитически, а оставшийся ряд будет сходиться гораздо быстрее, чем исходный.

Таковы основные «универсальные» рецепты для бесконечных рядов. Все остальные рецепты соответствуют тем или иным частным случаям.

11.1. Квазигеометрический ряд

Квазигеометрический ряд — это ряд, у которого асимптотика n-го члена есть $a_n \simeq bq^n$. Сходимость такого ряда линейна, т. е. количество верных

знаков пропорционально числу операций. Действительно, для геометрической прогрессии

$$S_N = \sum_{n=1}^{N} a_n = \frac{b(q^{N+1}-1)}{q-1},$$

поэтому остаточный член равен $bq^{N+1}/(1-q)$. Тем не менее, сходимость можно ускорить дополнительно. Именно, следует вычислять

$$S'_N = S_N - \frac{(S_N - S_{N-1})^2}{S_N - 2S_{N-1} + S_{N-2}}.$$

Для настоящей геометрической прогрессии получим

$$S'_N = \frac{b(q^{N+1}-1)}{q-1} - \frac{b}{q-1}\,\frac{(q^{N+1}-q^N)^2}{q^{N+1}-2q^N+q^{N-1}} = \\ = \frac{b(q^{N+1}-1)}{q-1} - \frac{b}{q-1}\,q^{N+1} = \frac{-b}{q-1}.$$

что совпадает с точным ответом для S_∞ при произвольном N.

Для квазигеометрической прогрессии величину S'_N мы полагаем текущим ответом для всего ряда, а разница между S'_N и S'_{N-1} есть оценка погрешности этого ответа. (Для геометрической прогрессии при оценке остаточного члена нет необходимости увеличивать верхний предел суммирования в полтора-два раза: остаточный член геометрической прогрессии порядка текущего члена a_N, точнее, порядка $a_{N+1}/(q-1)$.)

11.2. Знакопостоянный ряд

Знакопостоянные ряды гораздо хуже знакопеременных. Достаточно сравнить ряды $\sum(-1)^n/n^2$ и $\sum 1/n^2$. В первом случае конечная сумма

$$S_N = \sum_{n=1}^{N} a_n$$

приближает искомый ответ S_∞ с погрешностью порядка текущего члена, т. е. порядка $1/n^2$, что вполне приемлемо при численном счете. Во втором случае остаточный член $\sum\limits_{n=N+1}^{\infty} a_n \approx \int\limits_{N}^{\infty} a(x)\,dx = 1/N$, так что достичь

хорошей точности за конечное время невозможно. Более того, при суммировании большого количества слагаемых накопится ошибка округления, так что точного ответа не получится даже при очень длительном счете.

Если «универсальный» рецепт с вычитанием асимптотики не помогает, то можно попытаться применить интерполяцию. Именно, вычислять суммы $S_{N_1}, S_{N_2}, S_{N_3}, \ldots$ (здесь $N_i = N_0 c^i$, а c порядка полутора-двух); на каждом k-ом шаге проводить полиномиальную интерполяцию по k точкам (x_i, y_i), $i = 1, \ldots, k$ (здесь $x_i = 1/N_i$, $y_i = S_{N_i}$) в точку $x = 0$, соответствующую бесконечному верхнему пределу суммирования; результат интерполяции S'_k считается текущим ответом, а разница $|S'_k - S'_{k-1}|$ есть оценка его погрешности. К сожалению, для рядов интерполяционный алгоритм далеко не так эффективен, как для интегралов, тем не менее нередко он дает заметное ускорение.

Другая, довольно экзотическая, возможность состоит в применении следующей теоремы:

Теорема: знакопостоянный ряд в действительности является знакопеременным.

Заменим знакопостоянный ряд $\sum a_n$ на знакопеременный ряд $\sum(-)^{n+1} b_n$, где

$$b_n = a_n + 2a_{2n} + 4a_{4n} + 8a_{8n} + \cdots .$$

Убедимся, что суммы этих рядов равны. Коэффициент при a_n в ряде $\sum a_n$ равен 1. Найдем коэффициент при a_n в ряде $\sum(-)^{n+1} b_n$. Всякое n может быть представлено в виде $n = 2^p m$, где m — нечетное. Тогда коэффициенты при a_n будут таковы:

- коэффициенты от $b_{m2^p}, b_{m2^{p-1}}, \ldots b_{m2^1}$ будут равны $-1, -2, \ldots -2^{p-1}$;
- коэффициент от b_m будет равен $+2^p$.

Поскольку $1 + 2 + \cdots 2^{p-1} = 2^p - 1$, то сумма всех коэффициентов действительно равна 1, что и требовалось доказать.

К сожалению, как показывает опыт, этот изящный метод зачастую применить не удается. Дело в том, что каждый член нового знакопеременного ряда сам по себе вычисляется как сумма бесконечного ряда (хотя этот ряд, как правило, достаточно быстро сходится). Особенно осложняется дело в

том случае, когда члены старого ряда a_n вычисляются с помощью рекурсии, т. е. один за другим, а явной формулы для a_n с произвольным индексом n нет или эта формула слишком громоздкая.

11.3. Знакопеременный ряд

Обычно знакопеременный ряд, особенно если абсолютная величина членов ряда достаточно быстро убывает, сходится быстро. Более того, если абсолютная величина членов ряда убывает монотонно, то известен точный интервал, в котором лежит ответ, действительно: $S_\infty \in [S_{N-1}, S_N]$.

Однако сходимость знакопеременного ряда можно ускорить дополнительно, если воспользоваться алгоритмом Эйлера:

$$\sum_{m=1}^{\infty}(-)^m a_m = \sum_{m=1}^{k-1}(-)^m a_m + \sum_{r=1}^{\infty}\frac{(-)^{k+r}}{2^{r+1}}\Delta_{k,r}.$$

Здесь $\Delta_{k,r}$ — это разностная производная вперед:

$$\Delta_{k,0} = a_k \quad \Delta_{k,1} = a_{k+1} - a_k \quad \Delta_{k,2} = a_{k+2} - 2a_{k+1} + a_k \quad \cdots.$$

Явная формула имеет вид:

$$\Delta_{k,r} = \sum_{p=0}^{r} C_r^p (-)^{r-p} a_{k+p},$$

что соответствует рекуррентной формуле

$$\Delta_{k,r} = \Delta_{k+1,r-1} - \Delta_{k,r-1}.$$

Прежде всего проверим, что формула Эйлера правильна. Убедимся, что коэффициент при a_{k+p} в формуле Эйлера действительно равен $(-)^{k+p}$. Используем явное выражение для $\Delta_{k,r}$ во втором слагаемом формулы Эйлера:

$$\sum_{r=1}^{\infty}\frac{(-)^{k+r}}{2^{r+1}}\Delta_{k,r} = \sum_{r=1}^{\infty}\frac{(-)^{k+r}}{2^{r+1}}\sum_{p=0}^{r} C_r^p (-)^{r-p} a_{k+p}.$$

Следовательно, коэффициент при a_{k+p} равен

$$\sum_{p=r}^{\infty}\frac{(-)^{k+r}}{2^{r+1}} C_r^p (-)^{r-p} = (-)^{k+p}\sum_{p=r}^{\infty}\frac{1}{2^{r+1}} C_r^p.$$

Убедимся, что

$$\sum_{p=r}^{\infty} \frac{1}{2^{r+1}} C_r^p = 1.$$

Действительно, можно записать следующее разложение в степенной ряд:

$$\frac{1}{(1-x)^{p+1}} = \sum_{r=0}^{\infty} \frac{(p+r)!}{p!} \frac{x^r}{r!}.$$

Сдвигая индекс суммирования r вверх на p, получим

$$\sum_{r=p}^{\infty} \frac{r!}{p!(r-p)!} x^{r-p}(1-x)^{p+1} = 1,$$

откуда при $x = 1/2$ имеем

$$\sum_{r=p}^{\infty} C_r^p (1/2)^{r+1} = 1,$$

что и требовалось доказать.

Итак, формула правильная. Но возникает естественный вопрос: ради чего мы заменяем один знакопеременный ряд другим? Дело в том, что член нового ряда $\Delta_{k,r}$ убывает при росте r быстрее, чем член старого ряда a_{k+r}. Действительно, если асимптотика членов ряда a_k имеет вид $1/k$, то $a_{k+r} \approx 1/r$, в то же время величины $\Delta_{k,r}$

$$\Delta_{k,1} \approx \frac{1}{(k+1)} - \frac{1}{k} = -\frac{1}{k(k+1)},$$

$$\Delta_{k,2} \approx -\frac{1}{(k+1)(k+2)} + \frac{1}{k(k+1)} = \frac{2}{k(k+1)(k+2)},$$

$$\dots$$

$$\Delta_{k,r} \approx \frac{r!k!}{(k+r)!}$$

убывают с ростом r гораздо лучше, чем $1/r$. Вопрос о том, как именно выбрать точку k, в которой мы переходим от вычисления ряда «в лоб» к вычислению ряда из разностных производных $\Delta_{k,r}$, обсуждается ниже.

Для реализации алгоритма Эйлера на практике следует применить следующий трюк:

Алгоритм:

На $k+t$-ом шаге, т. е. когда мы вычислили члены ряда вплоть до a_{k+t}, мы храним не разностные производные «вперед»:

$$\Delta_{k,0} \quad \dots \quad \Delta_{k,t},$$

а, наоборот, разностные производные «назад», деленные на соответствующие степени двойки:

$$\begin{gathered}\Delta_{k+t,0} = a_{k+t}\\ \Delta_{k+t,-1}/2 = (a_{k+t} - a_{k+t-1})/2\\ \Delta_{k+t,-2}/4 = (a_{k+t} - 2a_{k+t-1} + a_{k+t-2})/4\\ \dots\\ \Delta_{k+t,-t}/2^t.\end{gathered}$$

При этом, как легко проверить, $\Delta_{k+t,-t} = \Delta_{k,t}$.

К этому моменту мы уже вычислили конечный отрезок ряда из формулы Эйлера:

$$S_{k,t} = \sum_{m=1}^{\infty}(-)^m a_m = \sum_{m=1}^{k-1}(-)^m a_m + \sum_{r=1}^{t}\frac{(-)^{k+r}}{2^{r+1}}\Delta_{k,r}.$$

На следующем шаге мы вычисляем a_{k+t+1}.

После этого проводятся вычисления:

$$\begin{gathered}\Delta_{k+t+1,0} = a_{k+t+1}\\ \Delta_{k+t+1,-1}/2 = (a_{k+t+1} - a_{k+t})/2 = (\Delta_{k+t+1,0} - \Delta_{k+t,0})/2\\ \Delta_{k+t+1,-2}/4 = (\Delta_{k+t+1,-1}/2 - \Delta_{k+t,-1}/2)/2\\ \Delta_{k+t+1,-3}/8 = (\Delta_{k+t+1,-2}/4 - \Delta_{k+t,-2}/4)/2\\ \dots\\ \Delta_{k+t+1,-t-1}/2^{t+1} = (\Delta_{k+t+1,-t}/2^t - \Delta_{k+t,-t}/2^t)/2.\end{gathered}$$

Если выполняется условие

$$|\Delta_{k+t+1,-t-1}|/2^{t+2} < |\Delta_{k+t,-t}|/2^{t+1},$$

т. е. очередное слагаемое в формуле Эйлера меньше предыдущего,

то мы просто продолжаем вычислять ряд по разностным производным:

$$S_{k,t} \to S_{k,t+1} = S_{k,t} + (-)^{k+t+1}\Delta_{k+t+1,-t-1}/2^{t+2}$$

и запоминаем еще одну новую разностную производную $\Delta_{k+t+1,-t-1}/2^{t+1}$.

В противном случае мы делаем вывод, что точка k выбрана неудачно и ее надо увеличить на единичку:

$$S_{k,t} \to S_{k+1,t} = S_{k,t} + (-)^{k+t+1}\Delta_{k+t+1,-t-1}/2^{t+1}.$$

В этом случае разностная производная $\Delta_{k+t+1,-t-1}/2^{t+1}$ оказывается «лишней», поэтому она не запоминается.

Таким образом, в алгоритме реализован автоматический выбор «точки перехода» k.

Чтобы обосновать изложенный алгоритм, достаточно доказать, что $S_{k+1,t} - S_{k,t} = (-)^{k+t+1}\Delta_{k+t+1,-t-1}/2^{t+1}$.
Действительно,

$$S_{k+1,t} - S_{k,t} = (-)^k a_k + (-)^{(k+1)}\Big[\Delta_{k+1,0}/2 - \Delta_{k+1,1}/4 +$$
$$+\Delta_{k+1,2}/8 - \Delta_{k+1,3}/16 + \cdots + (-)^t\Delta_{k+1,t}/2^{t+1}\Big] -$$
$$-(-)^{(k)}\Big[\Delta_{k,0}/2 - \Delta_{k,1}/4 + \Delta_{k,2}/8 -$$
$$-\Delta_{k,3}/16 + \cdots + (-)^t\Delta_{k,t}/2^{t+1}\Big].$$

Воспользуемся в первой сумме рекуррентной формулой для разностных производных $\Delta_{k+1,n} = \Delta_{k,n+1} + \Delta_{k,n}$:

$$S_{k+1,t} - S_{k,t} = (-)^k a_k + (-)^{(k+1)}\Big[\Delta_{k,1}/2 + \Delta_{k,0}/2 - \Delta_{k,2}/4 - \Delta_{k,1}/4 +$$
$$+\Delta_{k,3}/8 + \Delta_{k,2}/8 \cdots + (-)^t\Delta_{k,t+1}/2^{t+1} + (-)^t\Delta_{k+1,t}/2^{t+1}\Big] +$$
$$+(-)^{(k+1)}\Big[\Delta_{k,0}/2 - \Delta_{k,1}/4 + \Delta_{k,2}/8 -$$
$$-\Delta_{k,3}/16 + \cdots + (-)^t\Delta_{k,t}/2^{t+1}\Big].$$

После этого большая часть слагаемых сокращается и остается лишь

$$S_{k+1,t} - S_{k,t} = (-)^k a_k + (-)^{(k+1)}\Delta_{k,0} + (-)^{k+t+1}\Delta_{k,t+1}/2^{t+1} = \\ = (-)^{k+t+1}\Delta_{k,t+1}/2^{t+1}.$$

что и требовалось доказать.

Опыт показывает, что алгоритм Эйлера действительно может заметно ускорить сходимость знакопеременного ряда.

11.4. Цепные дроби

Цепная дробь — это конструкция следующего вида:

$$R_4 = \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4}}}}.$$

Обычно цепную дробь условно записывают в виде

$$R_4 = \frac{a_1}{b_1+}\ \frac{a_2}{b_2+}\ \frac{a_3}{b_3+}\ \frac{a_4}{b_4}.$$

Как правило, цепные дроби возникают при вычислении тех или иных высших трансцендентных функций. Если функция может быть представлена в виде цепной дроби, то этим обязательно надо воспользоваться, поскольку бесконечные цепные дроби обычно сходятся гораздо быстрее, чем ряды.

Разумеется, цепные дроби не следует вычислять в соответствии с явным выражением. Прямое вычисление цепной дроби не допускает рекурсии: если добавить к b_4 слагаемое a_5/b_5, то всю дробь придется пересчитывать заново.

Правильная рекурсивная процедура такова:

$$R_n = A_n/B_n$$
$$A_n = b_n A_{n-1} + a_n A_{n-2}$$
$$B_n = b_n B_{n-1} + a_n B_{n-2}.$$

Чтобы начать эту рекурсию, следует положить $A_0 = 0$, $B_0 = 1$, $A_1 = a_1$, $B_1 = b_1$.

Докажем (по индукции), что эта рекурсивная процедура действительно дает правильный ответ. Если выполняется соотношение

$$R_n = \frac{b_n A_{n-1} + a_n A_{n-2}}{b_n B_{n-1} + a_n B_{n-2}}, \qquad (*)$$

то ответ для R_{n+1} можно получить, просто заменив b_n на $b_n + a_{n+1}/b_{n+1}$. Следовательно,

$$R_{n+1} = \frac{(b_n + a_{n+1}/b_{n+1})A_{n-1} + a_n A_{n-2}}{(b_n + a_{n+1}/b_{n+1})B_{n-1} + a_n B_{n-2}} = \\ = \frac{a_{n+1}/b_{n+1} A_{n-1} + A_n}{a_{n+1}/b_{n+1} B_{n-1} + B_n} = \frac{a_{n+1}A_{n-1} + b_{n+1}A_n}{a_{n+1}B_{n-1} + b_{n+1}B_n},$$

но это выражение в точности совпадает с выражением $(*)$, в котором индекс n увеличен на единицу, что и требовалось доказать.

Единственная тонкость заключается в том, что в этом алгоритме есть опасность переполнения: при конечном R_n величины A_n и B_n могут одновременно устремиться к бесконечности (или, может быть, к нулю). Так что необходимо проверять их абсолютные величины на предмет переполнения (скажем, не превышают ли они 10^{50}) и «недополнения» (не становятся ли они меньше, чем 10^{-50}). В случае выхода за эти рамки следует поделить величины A_{n-1}, B_{n-1}, A_n и B_n на, скажем, $|B_n|$.

12. Системы линейных уравнений

12.1. Триангуляция

Как ни странно, если надо просто решить систему линейных уравнений (СЛУ) $n \times n$: $\hat{A}\vec{x} = \vec{b}$ для одной или нескольких правых частей $\vec{b}_i$, то самым простым и достаточно быстрым алгоритмом по-прежнему является тривиальная триангуляция. Если правых частей несколько ($i = 1 \ldots m$), то весь набор СЛУ можно записать как $\hat{A}\hat{x} = \hat{b}$, где матрицы $\hat{x}$ и $\hat{b}$ имеют размерность $n \times m$.

Алгоритм:

Сначала для 1-го столбца матрицы $\hat{A}$ $a_{i,1}$ мы ищем максимальный по модулю элемент, пусть это $a_{k,1}$.

Меняем местами 1-ю и k-ю строки матриц A и b.

Делим 1-ю строку матриц A и b на a_{11}.

Для всех $k = 2 \ldots n$

вычитаем из k-й строки матрицы A 1-ю строку, умноженную на a_{k1},

аналогично, из k-й строки матрицы b вычитаем 1-ю строку, также умноженную на a_{k1}.

В результате вся поддиагональная часть 1-го столбца занулится, а на диагонали стоит единица.

После этого забываем про 1-ю строку матриц A и b и про 1-й столбец матрицы A. Повторяем изложенные манипуляции для урезанной матрицы $(2 \ldots n) \times (2 \ldots n)$.

И так далее, пока матрица не кончится.

Довольно ясно, что каждый последующий этап никак не портит достигнутое на предыдущих этапах.

Результатом будет верхняя треугольная матрица с единицами на диагонали:

$$\begin{pmatrix} 1 & a_{12} & a_{13} & a_{14} & \ldots & a_{1n} \\ 0 & 1 & a_{23} & a_{24} & \ldots & a_{2n} \\ 0 & 0 & 1 & a_{34} & \ldots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 & a_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Теперь $\hat{x}$ легко найти обратной подстановкой: $x_{nk} = b_{nk}$ (для всех $k = 1 \ldots m$); после чего $x_{ik} = b_{ik} - \sum\limits_{j=i+1}^{n} a_{ij} x_{jk}$, где $i = n-1$, $n-2, \ldots, 1$ (также для всех $k = 1 \ldots m$).

Для несингулярных матриц этот алгоритм сбоев не дает. С точки зрения численного счета, сингулярная матрица — это вовсе не матрица с нулевым детерминантом. Более того, это не есть матрица с очень маленьким детерминантом. Ясно, что умножая всю матрицу как целое на, скажем, 0.1 (что нисколько не меняет ее свойств при решении СЛУ), мы уменьшаем ее детерминант в 10^n раз. Так что сингулярность мат-

рицы есть численная неотличимость одного (или нескольких) собственных значений матрицы от нуля на фоне остальных собственных значений. Т. е. сингулярность — это слишком большой перепад в собственных значениях, который делает ошибки округления при вычитании двух «больших» собственных значений больше, чем «маленькое» собственное значение.

Число операций при триангуляции пропорционально n^3, что, к сожалению, действительно очень много.

Заметим, что при триангуляции мы еще и вычислили детерминант матрицы A, поскольку детерминант есть произведение тех чисел a_{ii}, на которые мы делили, умноженное на $(-1)^s$, где s — число фактически выполненных перестановок. (Оно вполне может отличаться от $n-1$, т. к. максимальный элемент может уже стоять на диагонали, а в этом случае перестановка не понадобится.)

Кроме того, это есть рецепт нахождения матрицы, обратной к A, — достаточно взять $m = n$ и $\hat{b} = \hat{E}$.

Замечание:

Найденная таким образом A^{-1} ни в коем случае не является способом раз и навсегда решить СЛУ для данной матрицы A. Дело в том, что после триангуляции невязка СЛУ, т. е. компоненты $\hat{A}\vec{x} - \vec{b}$ будут порядка машинного нуля, т. е. (при 8-байтовых `double`) около 10^{-15}. (Мы полагаем, что матричные элементы A_{ij} и элементы правой части b_i порядка единицы, матрица регулярна и не слишком велика, скажем, 500×500.) Если же попытаться искать решение x как $x' = A^{-1}b$, то невязка $Ax' - b$ может оказаться порядка $10^{-8} \cdots 10^{-10}$, т. е. в $10^7 \cdots 10^5$ раз хуже. Самое удивительное, что при этом невязка при вычислении A^{-1} по-прежнему порядка машинного нуля: компоненты $AA^{-1} - E$ будут порядка 10^{-15}. Это связано с тем, что при численном счете $(AA^{-1})b$ не есть то же самое, что $A(A^{-1}b)$. Результат зависит от порядка суммирования: $(AA^{-1})b$ отличается от b не более чем на 10^{-15}, а $A(A^{-1}b)$ — уже на $10^{-8} \cdots 10^{-10}$.

12.2. LU-разложение

Это способ решить СЛУ для данной матрицы один раз и навсегда (впрок, для любых правых частей b, которые сейчас неизвестны, но потом возникнут).

Матрица $\hat{A}$ представляется в виде:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ L_{21} & 1 & 0 & \dots & 0 \\ L_{31} & L_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & L_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & \dots & U_{1n} \\ 0 & U_{22} & U_{23} & \dots & U_{2n} \\ 0 & 0 & U_{33} & \dots & U_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & U_{nn} \end{pmatrix}.$$

Очевидно, что любая СЛУ, записанная в таком виде, т. е. $Ax = LUx = y$, является уже решенной: если ввести обозначение z для решения системы $Lz = y$, то x является решением системы $Ux = z$. Обе системы треугольные, т. е. решаются подстановкой: $Lz = y$ — прямой, а $Ux = z$ — обратной. Действительно, поскольку $\sum_{i=1}^{k-1} L_{ki} z_i + z_k = y_k$, то z_1 определяется сразу, а все прочие z_k выражаются через ранее найденные z_j $(j < k)$. Аналогично, поскольку $\sum_{i=k+1}^{n} U_{ki} x_i + U_{kk} x_k = z_k$, то x_n определяется сразу, а прочие x_k находятся через ранее найденные x_j $(j > k)$.

Все уравнения, связывающие матричные элементы A с матричными элементами L и U, делятся на три группы:

1. Для наддиагональных элементов A:

$$A_{ik} = \sum_{j=1}^{i-1} L_{ij} U_{jk} + \underline{U_{ik}} \qquad (k > i).$$

   - Это соотношение позволяет определить элемент U_{ik} при условии, что мы знаем все элементы L, содержащиеся в столбцах $1 \dots k-1$, и, кроме того, все элементы k-го столбца U, лежащие выше U_{ik}, т. е. все U_{jk} при $j < i$.

2. Для диагональных элементов A:

$$A_{ii} = \sum_{j=1}^{i-1} L_{ij} U_{ji} + \underline{U_{ii}}.$$

   - Это соотношение позволяет определить диагональный элемент U_{ii} при тех же условиях, что и приведенные выше (этот случай можно было бы рассматривать как частный случай п. 1). Причина, по которой диагональный элемент рассматривается отдельно, станет очевидна впоследствии.

3. Для поддиагональных элементов A:

$$A_{ik} = \sum_{j=1}^{k-1} L_{ij}U_{jk} + \underline{L_{ik}}U_{kk} \qquad (k < i).$$

- Это соотношение позволяет определить элемент L_{ik} при условии, что уже известны все элементы U_{jk} (при $j \leqslant k$) для данного столбца, а также все элементы L, содержащиеся в столбцах $1 \ldots k-1$.

Из пп. 1–3 вытекает, что при использовании приведенных соотношений сначала для 1-го столбца для элементов $a_{11}, a_{21} \ldots a_{n1}$ в указанном порядке, далее для второго столбца в том же порядке $a_{12}, a_{22} \ldots a_{n2}$, и так далее, вплоть до n-го столбца в порядке $a_{1n}, a_{2n} \ldots a_{nn}$, условие их разрешимости каждый раз будет выполняться, т. е. мы сможем находить подчеркнутые величины из этих соотношений.

Единственная модификация, которую совершенно необходимо внести в эту последовательность действий, заключается в том, что придется переставлять строки в матрицах. При определении элементов L_{ik} нам приходится делить на диагональные элементы U_{kk}, т. е. мы должны позаботиться (в точности как при триангуляции), чтобы не делить на ноль (точнее, на накопившиеся ошибки округления). Довольно ясно, что на любом этапе манипуляций мы можем переставить строки в A и в L, никак не испортив того, что достигнуто на этот момент.

Сравнивая соотношения 2 и 3 мы видим, что в обоих случаях мы должны вычислить совершенно однотипные суммы:

$$A_{ik} - \sum_{j=1}^{k-1} L_{ij}U_{jk}.$$

При $i = k$ мы получим просто U_{kk}, а при $i > k$ результат надо разделить на U_{kk}, что даст L_{ik}. Следовательно, при определении диагонального элемента U_{kk} для k-го столбца надо начать с того, что вычислить все эти суммы для $k \leqslant i \leqslant n$ и выбрать наибольшую по абсолютной величине (скажем, при $i = i_0$). Это и будет будущий диагональный элемент U_{kk}. Чтобы он попал на диагональ, надо переставить строку i_0 с k-й строкой в матрицах A и L. Еще раз подчеркнем, что поскольку на данном этапе мы дошли только до k-го столбца, а $i_0 > k$, то эта перестановка ничего не испортит.

После перестановки, в результате которой U_{kk} попадет на диагональ, все остальные вычисленные суммы надо на него поделить, вычислив тем самым L_{ik}. При этом (в отличие от триангуляции) необходимо запомнить, какая именно перестановка проводилась для k-го столбца — ведь мы пока не располагаем правой частью СЛУ, чтобы в ней выполнять перестановки одновременно с перестановками в матрице.

Результатом всех этих манипуляций с учетом перестановок будет тождество

$$P_n P_{n-1} \dots P_2 P_1 A = LU,$$

где P_k — это оператор перестановки i_0-й и k-й строк, проведенной при работе с k-м столбцом (если вдруг оказалось $i_0 = k$, то $P_k = E$).

Так что перестановки строк в A приведут лишь к необходимости переставить в том же порядке и таким же образом элементы правой части СЛУ: $LUx = P_n P_{n-1} \dots P_2 P_1 y$.

И последнее маленькое замечание: разумеется, обе матрицы L и U помещаются в один массив.

12.3. Тридиагональные системы

В частном случае тридиагональной матрицы LU разложение приводит к линейному по размеру матрицы n алгоритму решения СЛУ. Действительно, тридиагональную матрицу можно разложить как

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & b_n \end{pmatrix} = \begin{pmatrix} v_1 & 0 & 0 & 0 & \dots & 0 \\ u_2 & v_2 & 0 & 0 & \dots & 0 \\ 0 & u_3 & v_3 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & v_n \end{pmatrix} \begin{pmatrix} 1 & w_1 & 0 & 0 & \dots & 0 \\ 0 & 1 & w_2 & 0 & \dots & 0 \\ 0 & 0 & 1 & w_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{pmatrix}.$$

При этом как процедура разложения, так и прямая и обратная подстановки потребуют числа операций, пропорционального всего лишь $O(n^1)$.

Соотношения, связывающие a_i, b_i, c_i с u_i, v_i, w_i, будут иметь вид:

$$b_k = u_k w_{k-1} + v_k \qquad a_k = u_k \qquad c_k = v_k * w_k$$

(здесь $w_0 \equiv 0$). Полагая $Lz = y$, $Ux = z$, получим

$$u_k z_{k-1} + v_k z_k = y_k \qquad x_k + w_k x_{k+1} = z_k$$

(здесь $z_0 \equiv 0$ и $x_{n+1} \equiv 0$).

Видно, что процедуры LU-разложения и прямой подстановки (вычисления z) можно проводить сверху вниз:

$$v_k = b_k - u_k w_{k-1} \qquad w_k = c_k/v_k \qquad u_k = a_k$$
$$z_k = (y_k - u_k z_{k-1})/v_k. \qquad (*)$$

Более того, их можно проводить одновременно, причем в результате мы должны получить значения z_i и w_i, необходимые для обратной подстановки. Это и составляет первый этап алгоритма. Значения u_i и v_i для обратной подстановки совершенно не нужны, т. е. можно не отводить память под массивы u_i и v_i; достаточно вычислять текущие значения u_k и v_k в ходе прямой подстановки. Именно поэтому мы поставили единички на диагональ в матрице U, а не в матрице L — это позволяет запоминать на один массив меньше.

Обратная подстановка

$$x_k = z_k - w_k x_{k+1} \qquad (**)$$

составляет второй этап алгоритма.

Разумеется, изложенная процедура есть в точности метод прогонки (прямая $(*)$ и обратная прогонки $(**)$), правда, в не очень стандартных обозначениях. Заметим, что в этой процедуре с неизбежностью происходит деление на диагональные элементы v_k. С этим приходится смириться, т. к. любая перестановка строк разрушает структуру матрицы. Следует вставить проверку равенства v_k машинному нулю, т. е. проверку на (почти) равенство b_k и $u_k w_{k-1}$. Если v_k (почти) зануляется, то придется пользоваться обычной триангуляцией, т. к. зануление v_k, вообще говоря, не означает сингулярности матрицы и перестановка строк вполне может спасти положение.

12.4. Экзотические частные случаи

Оба частных случая позволяют обратить матрицу не за N^3 операций, а за N^2 операций, что для больших матриц означает огромную экономию времени.

Матрица вида $A_{ij} = (X_j)^i$ или $\tilde{A}_{ij} = (X_i)^j$:

Алгоритм:

Прежде всего, вычислим коэффициенты C_k при степенях x^k (где $k = 0, \ldots, N$) в полиноме $P(x) = \prod_{k=1}^{N}(x - X_k)$. Это делается

тривиальной рекурсией за N^2 операций: начальные значения равны $C_0 = 1$, $C_{k>0} = 0$, а каждое умножение на очередное $(x - X_i)$ приводит к преобразованию: $C_m \to C_{m-1} - X_i C_m$. Разумеется, при этом преобразовании цикл по m должен идти сверху вниз, а не снизу вверх[7].

Далее, для каждого $j = 1, \ldots, N$ вычислим коэффициенты $C_k^{(j)}$ при степенях x^k (где $k = 0, \ldots, N-1$) в полиноме $P_j(x) = P(x)/(x - X_j)$. Это делается с помощью той же самой рекурсии, но запущенной в обратную сторону: $C_{N-1}^{(j)} = 1$ и $C_m^{(j)} = C_{m+1} + X_j C_{m+1}^{(j)}$ для $m = N-2, N-3, \ldots, 0$. Для каждого j имеем N операций, всего — N^2 операций.

Делим коэффициенты $C_k^{(j)}$ на $P_j(X_j)$, что также займет N^2 операций. Тем самым коэффициенты $C_k^{(j)}$ станут коэффициентами при степенях x^k для полинома

$$\tilde{P}_j(x) = P_j(x)/P_j(X_j) = \prod_{k \neq j} \frac{x - X_k}{X_j - X_k}.$$

Как ни странно, на этом задача обращения матрицы $A_{ij} = (X_j)^i$ и завершается, просто потому, что

$$\tilde{P}_j(X_i) = \delta_{ij} = \sum_{k=0}^{N-1} C_k^{(j)} (X_i)^k,$$

т. е. матрица, обратная к $A_{ij} = (X_i)^j$, есть в точности $(A^{-1})_{ij} = C_j^{(i)}$.

Во втором случае ($\tilde{A}_{ij} = (X_j)^i$) надо построить матрицу, обратную к транспонированной матрице $\tilde{A} = A^T$, т. е. транспонированную обратную матрицу $(\tilde{A}^{-1})_{ij} = C_i^{(j)}$.

К сожалению, эффективность этого метода значительно уступает его изяществу. При больших размерах матриц, как правило, оказывается, что

[7] Впрочем, в таких ситуациях разумнее не ломать голову, а заводить лишний вспомогательный массив. Для одномерных массивов машинной памяти, как правило, хватает, а добавлять в программу возможный источник ошибок ради (кажущейся) оптимальности неразумно.

той точности, которую обеспечивает `double`-арифметика, для этого алгоритма не хватает. Ситуация здесь такая же, как при вычислении полинома с известными коэффициентами при степенях x^k. Даже при правильном способе вычисления

$$(((\cdots(c_n * x + c_{n-1}) * x + \cdots) * x + c_2) * x + c_1) * x + c_0$$

мы вряд ли сумеем вычислить его с достаточной точностью, если значение x, во-первых, лежит между наименьшим и наибольшим корнем и, во-вторых, много больше единицы.

Для маленьких матриц метод работает, но выигрыш по скорости не так уж и велик — так что имеет смысл просто применять триангуляцию.

Матрица вида $A_{ij} = T_{i-j}$ или $A_{ij} = T_{j-i}$:

Такая матрица довольно часто встречается в теоретической физике: например, так выглядит гамильтониан парного взаимодействия в одномерной решеточной системе при условии трансляционной инвариантности.

В этом случае строится рекурсия, которая позволяет получить решение СЛУ размера $N \times N$ за N^2 операций. Основой рекурсии являются два семейства решений СЛУ, которые, на первый взгляд, никакого отношения к исходной СЛУ не имеют:

$$\sum_{j=1}^{M} T_{i-j} R_j^{(M)} = -T_i \qquad i = 1, \ldots, M; \qquad (*)$$

$$\sum_{s=1}^{M} T_{s-p} L_s^{(M)} = -T_{-p} \qquad p = 1, \ldots, M. \qquad (**)$$

Если мы располагаем решениями этих СЛУ размера $M \times M$, то мы можем построить решения для таких же СЛУ, но размером на единичку больше. Иными словами, мы можем совершить один шаг рекурсии $M \to M + 1$.

Действительно, если из первых M уравнений расширенной системы размера $(M+1) \times (M+1)$ для $R_j^{(M+1)}$ вычесть уравнения старой системы

размера $M \times M$ для $R_j^{(M)}$, то получится следующее:

$$\sum_{j=1}^{M} T_{i-j}(R_j^{(M+1)} - R_j^{(M)}) + T_{i-(M+1)} R_{M+1}^{(M+1)} = 0 \quad i = 1, \ldots, M.$$

Разделим эти уравнения на $R_{M+1}^{(M+1)}$ и заменим индексы $i = M + 1 - p$, $j = M + 1 - s$, тогда

$$\sum_{s=1}^{M} T_{s-p}\left[(R_{M+1-s}^{(M+1)} - R_{M+1-s}^{(M)})/R_{M+1}^{(M+1)}\right] = -T_{-p} \quad p = 1, \ldots, M.$$

Следовательно, выражение в квадратных скобках есть в точности решение СЛУ $(**)$:

$$\left[(R_{M+1-s}^{(M+1)} - R_{M+1-s}^{(M)})/R_{M+1}^{(M+1)}\right] = L_s^{(M)},$$

т. е. все $R_j^{(M+1)}$ при $j \leqslant M$ выражаются через $R_{M+1}^{(M+1)}$:

$$R_j^{(M+1)} = R_j^{(M)} + R_{M+1}^{(M+1)} L_{M+1-j}^{(M)}.$$

Само же $R_{M+1}^{(M+1)}$ легко найти из «лишнего» ($M + 1$-го) уравнения расширенной системы:

$$\sum_{j=1}^{M} T_{M+1-j} R_j^{(M+1)} + T_0 R_{M+1}^{(M+1)} = -T_{M+1}.$$

Подставляя сюда выражение для $R_j^{(M+1)}$, получим

$$\sum_{j=1}^{M} T_{M+1-j}(R_j^{(M)} + R_{M+1}^{(M+1)} L_{M+1-j}^{(M)}) + T_0 R_{M+1}^{(M+1)} = -T_{M+1},$$

откуда

$$R_{M+1}^{(M+1)} = \frac{-T_{M+1} - \sum_{j=1}^{M} T_{M+1-j} R_j^{(M)}}{T_0 + \sum_{j=1}^{M} T_{M+1-j} L_{M+1-j}^{(M)}}. \qquad (\star)$$

С помощью совершенно аналогичных рассуждений легко убедиться, что

$$L_s^{(M+1)} = L_s^{(M)} + L_{M+1}^{(M+1)} R_{M+1-s}^{(M)}$$

$$L_{M+1}^{(M+1)} = \frac{-T_{-(M+1)} - \sum_{s=1}^{M} T_{s-(M+1)} L_s^{(M)}}{T_0 + \sum_{s=1}^{M} T_{s-(M+1)} R_{M+1-s}^{(M)}}. \qquad (\star\star)$$

Итак, мы действительно можем построить решение двух СЛУ $(\star)$ и $(\star\star)$ произвольной размерности $M \times M$ за M^2 операций с помощью рекурсивной процедуры. На каждом шаге мы вычисляем новые значения $R_j^{(M)}$ и $L_s^{(M)}$. Чтобы начать рекурсию, достаточно начать с «системы уравнений» $M \times M$ при $M = 1$.

Интересно, что в случае симметричной матрицы (т. е. в случае $T_{-i}{=}T_i$) обе системы совпадают между собой, так что $L_i^{(M)} = R_i^{(M)}$.

Теперь обратимся к нашей исходной задаче. Совершенно ясно, что мы можем тем же рекурсивным способом получить решение СЛУ произвольной размерности $M \times M$ при произвольной правой части b_i:

$$\sum_{j=1}^{M} T_{i-j} x_j^{(M)} = b_i \quad i = 1, \ldots, M,$$

или для транспонированной матрицы при произвольной правой части c_{-p}:

$$\sum_{s=1}^{M} T_{s-p} y_s^{(M)} = c_{-p} \quad p = 1, \ldots M$$

(довольно нелепое обозначение c_{-p} применено здесь исключительно для единообразия в индексации). Влияние правой части на рекурсивную процедуру с очевидностью сводится лишь к изменению числителя в соотношениях $(\star)$ и $(\star\star)$.

Чтобы найти $x_j^{(M+1)}$, достаточно дополнить рекурсию вычислением:

$$x_j^{(M+1)} = x_j^{(M)} + x_{M+1}^{(M+1)} L_{M+1-j}^{(M)},$$

$$x_{M+1}^{(M+1)} = \frac{b_{M+1} - \sum_{j=1}^{M} T_{M+1-j} x_j^{(M)}}{T_0 + \sum_{j=1}^{M} T_{M+1-j} L_{M+1-j}^{(M)}}.$$

Соответственно, если нужно найти $y_s^{(M+1)}$, достаточно дополнить рекурсию вычислением:

$$y_s^{(M+1)} = y_s^{(M)} + y_{M+1}^{(M+1)} R_{M+1-s}^{(M)},$$

$$y_{M+1}^{(M+1)} = \frac{c_{-(M+1)} - \sum_{s=1}^{M} T_{s-(M+1)} y_s^{(M)}}{T_0 + \sum_{s=1}^{M} T_{s-(M+1)} R_{M+1-s}^{(M)}}.$$

Единственным существенным замечанием к изложенному рецепту является необходимость проверки деления на нуль (или почти нуль) в соотношениях $(\star)$ и $(\star\star)$. Как обычно, появление нуля в знаменателе, вообще говоря, не означает того, что матрица сингулярна. Скорее всего, это означает лишь крах изложенного алгоритма. Переставлять строки произвольным образом мы не можем, т. к. это разрушит структуру матрицы. Единственное, что можно попробовать сделать в этом случае, — это перенумеровать все индексы в обратном порядке, т. е. поменять местами начало и конец матрицы. Если это не поможет, придется смириться и вернуться к триангуляции.

Интересно, что совершенно аналогичную рекурсивную процедуру можно построить для матрицы, которая не может быть представлена в виде $A_{ij} = T_{i-j}$, но состоит из, скажем, четырех блоков, каждый из которых может быть представлен в таком виде. Например:

$$\begin{pmatrix} K_0 & K_{-1} & K_{-2} & Q_0 & Q_{-1} & Q_{-2} \\ K_1 & K_0 & K_{-1} & Q_1 & Q_0 & Q_{-1} \\ K_2 & K_1 & K_0 & Q_2 & Q_1 & Q_0 \\ M_0 & M_{-1} & M_{-2} & N_0 & N_{-1} & N_{-2} \\ M_1 & M_0 & M_{-1} & N_1 & N_0 & N_{-1} \\ M_2 & M_1 & M_0 & N_2 & N_1 & N_0 \end{pmatrix}.$$

Разумеется, процедура рекурсии будет гораздо сложнее (размеры блоков придется наращивать по очереди, т. е. придется различать «четные» и «нечетные» шаги рекурсии, кроме того, придется ввести 16 вспомогательных СЛУ вместо двух), но идея метода в точности такая же, и ее довольно легко реализовать.

12.5. Обращение слегка модифицированной матрицы

Легко проверить, что для любых трех матриц: матрицы A размерности $N \times N$, матрицы L размерности $N \times M$ и матрицы R размерности $M \times N$ — выполняется соотношение:

$$\Big(A - LR\Big)^{-1} = A^{-1} + A^{-1}L\Big(1 - RA^{-1}L\Big)^{-1}RA^{-1}.$$

Действительно,

$$\begin{aligned}\Big(A - LR\Big)^{-1} &= \Big(A\big[1 - A^{-1}LR\big]\Big)^{-1} = \Big(\big[1 - A^{-1}LR\big]^{-1}\Big)A^{-1} = \\ &= \Big(1 + A^{-1}LR + A^{-1}LR\,A^{-1}LR + \\ &\qquad + A^{-1}LR\,A^{-1}LR\,A^{-1}LR + \cdots\Big)A^{-1} = \\ &= A^{-1} + A^{-1}L\Big(1 + RA^{-1}L + RA^{-1}L\,RA^{-1}L + \cdots\Big)RA^{-1} = \\ &= A^{-1} + A^{-1}L\Big(1 - RA^{-1}L\Big)^{-1}RA^{-1},\end{aligned}$$

что и требовалось доказать.

Практическое применение этого соотношения очень простое: если $M \ll N$ и мы уже вычислили A^{-1}, то для обращения слегка модифицированной матрицы $(A - LR)$ достаточно обратить матрицу $(1 - RA^{-1}L)$ размерности $M \times M$, что делается гораздо быстрее, чем обращение матрицы $N \times N$. На практике, как правило, модифицируется только один элемент матрицы A (или только один столбец, или только одна строка). В этом случае $M = 1$, т. е. вообще никаких матриц обращать не надо. Действительно, если модифицируется один элемент матрицы $A_{ij} \to A_{ij} - \Delta a$, то $L_{k1} = \delta_{ki}\Delta a$, $R_{1k} = \delta_{kj}$. Аналогичные соотношения легко выписать, если модифицируется только одна строка или только один столбец.

13. Быстрое преобразование Фурье

13.1. Алгоритм FFT

Алгоритм FFT (Fast Fourier Transformation) используется, когда необходимо провести дискретное преобразование Фурье того или иного массива

с данными. Например, весь численный Фурье-анализ по необходимости использует именно дискретное преобразование Фурье.

$$b_k = \sum_{n=0}^{N-1} \frac{\exp(2\pi ikn/N)}{\sqrt{N}} a_n \qquad k = 0, \ldots, N-1,$$

$$a_n = \sum_{k=0}^{N-1} \frac{\exp(-2\pi ikn/N)}{\sqrt{N}} b_k \qquad n = 0, \ldots, N-1.$$

Если выполнять это Фурье-преобразование так, как это здесь написано, то понадобится N^2 операций. В действительности можно сильно сократить процедуру. Забудем пока про нормировку $1/\sqrt{N}$. Тогда

$$b_k = \sum_{m=0}^{N/2-1} e^{2\pi ik(2m)/N} a_{2m} + e^{2\pi ik/N} \sum_{m=0}^{N/2-1} e^{2\pi ik(2m)/N} a_{2m+1} = \\ = \sum_{m=0}^{N/2-1} e^{2\pi ikm/(N/2)} a_{2m} + e^{2\pi ik/N} \sum_{m=0}^{N/2-1} e^{2\pi ikm/(N/2)} a_{2m+1}. \qquad (*)$$

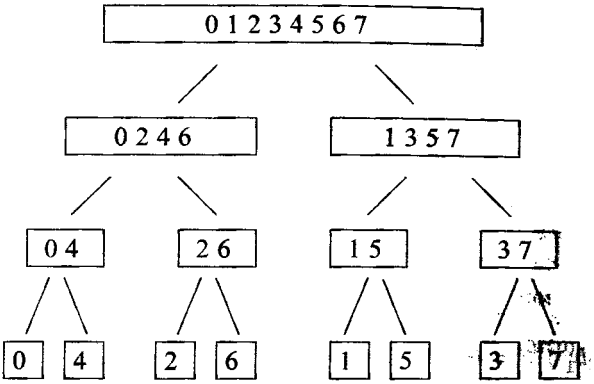
Тем самым количество работы сокращено вдвое: достаточно найти дискретные Фурье-образы «четной» (a_{2m}) и «нечетной» (a_{2m+1}) половин нашего массива при $k = 0 \ldots N/2-1$. Это потребует $2(N/2)^2 = N^2/2$ операций. Затем надо воспользоваться соотношением $(*)$, причем при $k < N/2$ им пользуются непосредственно, а при $k \geqslant N/2$

$$\sum_{m=0}^{N/2-1} \exp(2\pi ikm/(N/2)) f_m = \sum_{m=0}^{N/2-1} \exp(2\pi i(k-N/2)m/(N/2)) f_m$$

как при $f_m = a_{2m}$, так и при $f_m = a_{2m+1}$.

Довольно ясно, что на этом останавливаться нельзя, и имеет смысл еще раз поделить уже каждую из половин (как четную, так и нечетную) снова пополам. Тем самым возникает необходимость всегда иметь дело с массивами длиной $N = 2^L$, где L — некоторое целое число. Для экспериментаторов это означало бы необходимость дополнять фактически полученные данные нулями до ближайшего $N = 2^L$, но при численном счете в теоретической физике (как правило) может быть взят любой удобный размер массива — в данном случае $N = 2^L$.

Изложим окончательный алгоритм на примере $N = 8$. После трехкратного деления на 2



мы получим массивы единичной длины, для которых Фурье-образ совпадает с самим исходным массивом. Из картинки вытекает, что первым делом надо переставить элементы исходного массива 0 1 2 3 4 5 6 7 в том порядке, в котором они стоят в последней строчке — 0 4 2 6 1 5 3 7. Тогда элементы уже сгруппированы в правильном порядке для последовательного применения формулы (*) (04 и 26 и 15 и 37; 0246 и 1357). Упорядочение идет по очень простому принципу: при первом делении пополам в первую группу попадают четные числа (последний бит — 0), во вторую группу попадают нечетные числа (последний бит — 1), при последующем делении в первую группу попадают числа, являющиеся четными после деления пополам (предпоследний бит — 0), а во вторую группу попадают числа, являющиеся нечетными после деления пополам (предпоследний бит — 1), и т. д. Иными словами, упорядочить надо числа с битами, переставленными в обратном порядке:

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

0	4	2	6	1	5	3	7
000	100	010	110	001	101	011	111

Заметим, что пары чисел (41 и 26) просто меняются местами. Теперь применяем формулу $(*)$ для перехода от последнего к предпоследнему слою. В группе 04

$$b_0^{(04)} = a_0 + \exp(2\pi i 0/2)a_4 = a_0 + a_4$$
$$b_1^{(04)} = a_0 + \exp(2\pi i 1/2)a_4 = a_0 - a_4.$$

То же самое для групп 26, 15, 37. Разумеется, самым разумным способом размещения по элементам массива будет:

$$b_0^{(04)}\ b_1^{(04)}\quad b_0^{(26)}\ b_1^{(26)}\quad b_0^{(15)}\ b_1^{(15)}\quad b_0^{(37)}\ b_1^{(37)}.$$

Далее,

$$b_0^{(0246)} = b_0^{(04)} + \exp(2\pi i 0/4)b_0^{(26)} = a_0 + a_2 + a_4 + a_6$$
$$b_1^{(0246)} = b_1^{(04)} + \exp(2\pi i 1/4)b_1^{(26)} = a_0 + ia_2 - a_4 - ia_6$$
$$b_2^{(0246)} = b_2^{(04)} + \exp(2\pi i 2/4)b_2^{(26)} = a_0 - a_2 + a_4 - a_6$$
$$b_3^{(0246)} = b_3^{(04)} + \exp(2\pi i 3/4)b_3^{(26)} = a_0 - ia_2 + a_4 + ia_6,$$

то же для $1, 3, 5, 7$.

Естественный способ размещения в массиве:

$$b_0^{(0246)}\ b_1^{(0246)}\ b_2^{(0246)}\ b_3^{(0246)}\qquad b_0^{(1357)}\ b_1^{(1357)}\ b_2^{(1357)}\ b_3^{(1357)}.$$

И, наконец,

$$b_k = b_k^{(0246)} + \exp(2\pi ik/8)b_k^{(1357)} \qquad k = 0,\ 1,\ 2,\ 3$$
$$b_k = b_{k-4}^{(0246)} + \exp(2\pi ik/8)b_{k-4}^{(1357)} \qquad k = 4,\ 5,\ 6,\ 7.$$

Легко проверить, что эти выражения действительно дают правильные дискретные Фурье-образы.

Алгоритм для обратного преобразования Фурье отличается только знаком в показателе экспоненты.

Вообще говоря, этот алгоритм может быть применен не только по основанию 2, но и по основаниям 3, 5, 7 и т. д. Тем самым, для некоторого массива длины $N = 2^5 * 3^2 * 11^3$ можно было бы применить деления на 11, на 3 и т. д. Лучше, однако, этого не делать.

13.2. Замечания о дискретном преобразовании Фурье

Численный Фурье-анализ:

Если есть массив данных $a_0, \ldots, a_N$, который в некотором смысле изображает непрерывную функцию $a(x)$ ($a_i = a(i*dx)$) на некотором интервале, то его дискретное преобразование Фурье — массив $b_0, \ldots, b_N$ в некотором смысле изображает $b(k)$ — непрерывный Фурье-образ функции $a(x)$. Однако при этом необходимо помнить о двух вещах:

Во-первых, очень ограниченный набор дискретных волновых чисел k может претендовать на то, чтобы изображать сколько-нибудь гладкую «непрерывную» синусоидальную функцию на решетке a_i. В частности, $k = N/6$ и $k = 5N/6$ соответствуют бегущим влево и вправо волнам, период которых составляет 6 узлов решетки. Ясно, что говорить о «гладкой» синусоиде с меньшим периодом бессмысленно. Итак, только дискретные волновые числа $k = 0, \ldots, N/6$ и $k = 5N/6, \ldots, N-1$ могут претендовать на соответствие с непрерывным Фурье-образом $b(k)$.

Во-вторых, мы рассматриваем нашу функцию $a(x)$ на интервале от 0 до $L{=}N*dx$. Иными словами, мы вычислили вовсе не Фурье-образ $a(x)$, а Фурье-образ произведения $\tilde{a}(x) = a(x)\theta(0 < x < L)$. (Здесь $\theta(0 < x < L) = 1$ при $0 < x < L$ и $\theta(0 < x < L) = 0$ для всех остальных x.) Этот Фурье-образ равен свертке Фурье-образа $a(x)$ и Фурье-образа $\theta(0 < x < L)$:

$$\tilde{b}(k) \sim \int dp\, \theta(p-k)b(k) \qquad \theta(k) \sim (\exp(ikL) - 1)/ik.$$

Величина $|\theta(p-k)|^2 \sim \Big(\sin\big((p-k)L/2\big)/(p-k)\Big)^2$, разумеется, имеет пик при $p \approx k$, ширина этого пика порядка $2\pi/L$, что и составляет теоретически возможное разрешение для волнового числа при измерении на интервале $0 < x < L$. Конечно, основной вклад в интеграл дает именно эта окрестность, но и вкладом от других областей пренебречь нельзя. Дело в том, что $|\theta(p-k)|^2$ не так уж быстро убывает с ростом $|p-k|$, и связано это, разумеется, со скачками в $\theta(0 < x < L)$. Поэтому для данного k_0 величина $\tilde{b}(k_0)$ не вполне совпадает с $b(k_0)$ — в нее дают заметный вклад значения $b(k)$ с k, далекими от k_0.

Нам бы хотелось, чтобы $\tilde{b}(k_0)$ было по возможности ближе к $b(k_0)$. Для этого можно несколько пожертвовать разрешением — слегка увеличить ширину пика, но зато заметно ускорить убывание при больших $|p-k|$. Для

этого необходимо заменить «резкую» функцию включения $\theta(0 < x < L)$ на плавную $\tilde{\theta}(0 < x < L)$, Фурье-образ которой будет убывать гораздо быстрее. Совершенно аналогичное рассуждение уже было приведено в разделе «Фурье-интерполяция», там же приведен один из возможных примеров такой гладкой функции включения — с гиперболическими тангенсами. Другая возможная функция включения — гауссова шляпка с центром в точке $x = L/2$. Ее ширина, с одной стороны, должна быть максимально возможной (чтобы не ухудшать разрешение), но при этом при $x = 0$ и $x = L$ она должна обращаться в (почти) ноль.

Вычисление автокорреляций и сверток:

При вычислении автокорреляций и сверток

$$\bar{a}_n = \sum_{i=0}^{N-1} a_{n+i} a'_i \qquad \tilde{a}_n = \sum_{i=0}^{N-1} a_{n-i} a'_i$$

в лоб количество операций пропорционально N^2. Если оба массива удовлетворяют условию периодичности, т. е. $a_{N+i} = a_i$, то использование FFT позволяет свести количество операций к $N \log N$[8]:

$$\bar{b}_k = b_k * b'_{-k}/\sqrt{N} \qquad \tilde{b}_k = b_k * b'_k/\sqrt{N}.$$

Иными словами, следует выполнить дискретное преобразование Фурье исходных массивов, перемножить соответствующие компоненты, а затем совершить обратное преобразование Фурье. Необходимость вычисления сверток встречается довольно часто (см., например, раздел «Арифметика произвольной точности»).

Решение СЛУ особого вида:

Если необходимо решить СЛУ

$$\sum_{i=0}^{N-1} A_{i-j} x_j = b_i \qquad i = 0, \ldots, N-1,$$

где коэффициенты A_i удовлетворяют условию периодичности $A_i = A_{i+N}$, то она также решается за $N \log N$ операций. Фурье-образы A_i, x_i и b_i

[8] Если массивы не удовлетворяют условию периодичности, то их можно поместить в массивы вдвое большего размера, заполнив «лишние» элементы нулями. Тем самым условие периодичности (с периодом $2N$) будет выполнено, а выигрыш по скорости мы все равно получим (хотя FFT придется применять к более длинным массивам).

связаны соотношением

$$A_k * x_k/\sqrt{N} = b_k,$$

что и составляет решение задачи.

14. Задача на СВ и СЗ

Постановка задачи такова: для данной симметричной (или эрмитовой) матрицы $\hat{A}$ надо найти либо только собственные значения (СЗ), либо собственные значения и соответствующие собственные вектора (СВ). Алгоритмы, которые будут изложены, могут применяться как к действительным симметричным матрицам, так и к комплексным эрмитовым. Опыт показывает, что в большинстве реальных физических задач можно так или иначе свести эрмитову матрицу к симметричной. Это экономит память (матрица занимает вдвое меньше места) и заметно ускоряет счет (нет необходимости прибегать к комплексной арифметике).

Задача о поиске левых и правых собственных векторов неэрмитовой матрицы ни разу не встречалась авторам в их практической работе. В принципе, такая задача может возникнуть, например, когда необходимо обратить почти сингулярную неэрмитову матрицу. Тем не менее, мы уклонимся от подробного разбора случая неэрмитовой матрицы.

Основные алгоритмы поиска СВ и СЗ с необходимостью опираются на операцию, которую можно назвать «элементарным вращением» в плоскости (k, q). Эта операция реализуется унитарной (в случае вещественной матрицы A – ортогональной) матрицей U, все элементы которой совпадают с элементами единичной матрицы, за исключением четырех: U_{kk}, U_{kq}, U_{qk}, U_{qq}, причем эти 4 элемента образуют унитарную матрицу 2×2. Довольно ясно, что в результате унитарного вращения $A \to U^+AU$ в матрице A будут затронуты лишь элементы, принадлежащие k-му и q-му столбцам и k-й и q-й строкам:

$$A_{ij} \to A_{ij}$$

$$A_{ik} \to A_{ik}U_{kk} + A_{iq}U_{qk} \qquad A_{iq} \to A_{ik}U_{kq} + A_{iq}U_{qq}$$

$$A_{ki} \to U^+_{kk}A_{ki} + U^+_{kq}A_{qi} \qquad A_{qi} \to U^+_{qk}A_{ki} + U^+_{qq}A_{qi}$$

(здесь i и j не совпадают ни с k, ни с q). Кроме того, разумеется,

$$\begin{pmatrix} A_{kk} & A_{kq} \\ A_{qk} & A_{qq} \end{pmatrix} \to \begin{pmatrix} U^+_{kk} & U^+_{kq} \\ U^+_{qk} & U^+_{qq} \end{pmatrix} \begin{pmatrix} A_{kk} & A_{kq} \\ A_{qk} & A_{qq} \end{pmatrix} \begin{pmatrix} U_{kk} & U_{kq} \\ U_{qk} & U_{qq} \end{pmatrix}.$$

Поскольку унитарное вращение сохраняет эрмитовость, то нет никакой необходимости проводить выкладки для, скажем, поддиагональных матричных элементов A. Причем дело не только в экономии времени. Алгоритмы довольно чувствительны к эрмитовости матрицы A, и если за ней специально не следить, то нарушения эрмитовости из-за ошибок округления будут довольно быстро накапливаться и приведут к значительной погрешности. Погрешность будет гораздо меньше, если эрмитовость соблюдать принудительно — т. е. не вычислять A_{ji}, а сразу полагать его равным A^*_{ij}.

Большинство алгоритмов сводятся к тому, что цепочка тех или иных последовательных элементарных вращений $U_1, U_2, \ldots U_M$ превращает матрицу A в диагональную. Разумеется, диагональность достигается с той или иной точностью ϵ, эта точность, в свою очередь, определяет погрешность в найденных СВ и СЗ. При этом если СВ нас не интересуют, то нет никакой необходимости запоминать (накапливать) ту итоговую матрицу поворота $U_{tot} = U_1 U_2 \cdots U_M$, которая диагонализует A — это позволит заметно уменьшить число операций. Если же нам нужны еще и СВ, то при каждом очередном элементарном вращении U_l необходимо уже накопленную матрицу $U_1 U_2 \cdots U_{l-1}$ домножить справа на U_l (напомним, что U_l — элементарное вращение, так что это домножение занимает не N^3 операций, как при умножении произвольных матриц, а всего N операций).

В итоге мы получим $U^+_{tot} A U_{tot} = \tilde{A}$, где $\tilde{A}$ — диагональна с некоторой точностью ϵ. Следовательно, $\tilde{A}_{kk}$ есть k-е собственное значение матрицы A, а k-й столбец матрицы U, т. е. $x^{(k)}_i = U_{ik}$, есть соответствующий собственный вектор. Легко проверить, что точность в определении СЗ будет пропорциональна вовсе не ϵ, а ϵ^2. Поэтому если нужен только спектр, то не следует заказывать слишком маленькое ϵ — это приведет к ухудшению точности из-за накопления ошибок округления при излишне большом числе операций. В то же время следует помнить, что погрешность в СВ пропорциональна ϵ.

14.1. Метод Якоби

Алгоритм:

Для всех недиагональных элементов матрицы A подряд, т. е. $A_{12}, A_{13}, \ldots, A_{1N}, A_{23}, \ldots, A_{2N}, \ldots, A_{N-1\,N}$ проделывают следующее:

Если $|A_{ik}| < \epsilon$, то переходят к следующему элементу.

В противном случае выполняют элементарное вращение в плоскости (ik), которое занулит A_{ik}. (Разумеется, это вращение испортит результат предыдущего вращения. Элемент, обращенный в нуль предыдущим вращением, вновь станет отличен от нуля. С этим необходимо смириться.)

Если при очередном проходе всей матрицы ни одного вращения не было, то процедура прекращается. Как правило, процедура сходится не более чем за десяток проходов.

Алгоритм сходится по очень простой причине: при каждом элементарном вращении в плоскости (ik) сумма квадратов модулей всех внедиагональных элементов убывает в точности на $2|A_{ik}|^2$. Действительно, меняются только элементы i-й и k-й строк, а также элементы i-го и k-го столбцов. Элементы A_{ik} и A_{ki} тождественно зануляются, а остальные элементы попарно подвергаются унитарному вращению:

$$A_{si} \to A_{si}U_{ii} + A_{sk}U_{ki} \qquad A_{sk} \to A_{si}U_{ik} + A_{sk}U_{kk}$$
$$A_{is} \to U^+_{ii}A_{is} + U^+_{ik}A_{ks} \qquad A_{ks} \to U^+_{ki}A_{is} + U^+_{kk}A_{ks}$$

$(s \neq i, k)$, т. е. сумма квадратов их модулей не меняется.

Можно встретить вариацию алгоритма Якоби, в которой поворачивают не все внедиагональные элементы подряд, а находят максимальный по модулю внедиагональный элемент и зануляют его элементарным вращением, затем вновь находят максимальный элемент и зануляют его, и т. д. Ясно, что при этом «коэффициент полезного действия» каждого отдельного вращения возрастает — сумма квадратов модулей внедиагональных элементов при каждом повороте убывает на максимально возможную величину. Но чтобы найти этот максимальный элемент, надо каждый раз пробежать всю матрицу. Опыт показывает, что этот подход может как ускорить, так и замедлить сходимость — все зависит и от вида исходной матрицы, и от ее размерности, и от соотношения скоростей сложения, умножения и извлечения квадратного корня на данной машине. Так что дать универсальную рекомендацию в данном случае невозможно.

Следует иметь в виду, что метод Якоби эффективен при не очень больших размерах матрицы — до $N \approx 50$, максимум до $N \approx 100$. Как обычно, дело не только в том, что метод не будет сходиться за конечное время, дело еще и в излишне большом количестве операций, которое приведет к ошиб-

кам округления и потере точности. Так что для больших матриц необходимо применять алгоритм, изложенный в следующем разделе.

Последнее, чисто техническое, замечание касается параметров элементарного вращения, зануляющего элемент A_{ik}. Довольно ясно, что это просто унитарное преобразование, диагонализующее эрмитову матрицу

$$\mathcal{A} = \begin{pmatrix} A_{ii} & A_{ik} \\ A_{ki} & A_{kk} \end{pmatrix} = a_0 + \vec{a} \cdot \vec{\sigma},$$

где σ_i — матрицы Паули, $a_0 = \mathrm{Tr}(\mathcal{A})/2$, $a_i = \mathrm{Tr}(\mathcal{A}\sigma_i)/2$. Иными словами, это задача о диагонализации оператора проекции спина $1/2$ на ось $\vec{n} = \vec{a}/|\vec{a}| = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$. Мы ищем такой унитарный поворот, после которого проекция спина на ось z будет иметь определенное значение. Эта задача допускает два решения — мы можем повернуться на угол $\alpha < \pi/2$ и на угол $\pi - \alpha$, больший, чем $\pi/2$. С формальной точки зрения, никакой разницы нет, но, как легко понять, ошибки округления при повороте на малый угол α будут гораздо меньше, чем при повороте на угол $\pi - \alpha$ (т. е. на почти 180°). Так что унитарную матрицу

$$\mathcal{U} = \begin{pmatrix} U_{ii} & U_{ik} \\ U_{ki} & U_{kk} \end{pmatrix}.$$

которая в любом случае состоит из СВ оператора проекции спина на ось $\vec{n}$:

$$|\psi^+\rangle = \begin{pmatrix} \cos(\theta/2) \\ \sin(\theta/2)\, e^{i\varphi} \end{pmatrix} \qquad |\psi^-\rangle = \begin{pmatrix} -\sin(\theta/2)\, e^{-i\varphi} \\ \cos(\theta/2) \end{pmatrix}.$$

необходимо сооружать так, чтобы она сохраняла знак проекции спина на ось z. Иными словами, при $a_3 > 0$ следует брать

$$\mathcal{U} = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2)\, e^{-i\varphi} \\ \sin(\theta/2)\, e^{i\varphi} & \cos(\theta/2) \end{pmatrix},$$

а при $a_3 < 0$ — наоборот,

$$\mathcal{U} = \begin{pmatrix} \sin(\theta/2) & \cos(\theta/2)\, e^{-i\varphi} \\ -\cos(\theta/2)\, e^{i\varphi} & \sin(\theta/2) \end{pmatrix}.$$

Выпишем для полноты явные выражения для $\cos(\theta/2)$, $\sin(\theta/2)$, $\cos\varphi$, $\sin\varphi$:

$$\cos(\theta/2) = \sqrt{\frac{1}{2}\left(1 + \frac{a_3}{|\vec{a}|}\right)} \qquad \sin(\theta/2) = \sqrt{\frac{1}{2}\left(1 - \frac{a_3}{|\vec{a}|}\right)}$$

$$\cos\varphi = \frac{a_1}{\sqrt{a_1^2 + a_2^2}} \qquad \sin\varphi = \frac{a_2}{\sqrt{a_1^2 + a_2^2}},$$

где $a_3 = (A_{ii} - A_{kk})/2$, $a_1 = \mathrm{Re}A_{ik}$, $a_2 = -\mathrm{Im}A_{ik}$. Из выражений для $\cos(\theta/2)$ и $\sin(\theta/2)$ очевидно, что учет знака a_3 действительно позволяет достигнуть максимальной близости $\mathcal{U}$ к единичной матрице, т. е. «минимизировать» вращение.

14.2. Алгоритм LQ (он же алгоритм QR)

Большинство элементарных вращений в этом разделе будет определяться не условием диагонализации той или иной матрицы, а линейным условием на элементы матрицы:

$$aU_{ii} + bU_{ki} = 0. \qquad (*)$$

Выпишем один раз явные выражения, вытекающие из $(*)$, а в дальнейшем будем ограничиваться соотношением типа $(*)$:

$$\begin{pmatrix} U_{ii} & U_{ik} \\ U_{ki} & U_{kk} \end{pmatrix} = \frac{1}{\sqrt{|a|^2 + |b|^2}} \begin{pmatrix} b & a^* \\ -a & b^* \end{pmatrix}.$$

Алгоритм состоит из двух этапов:

1-й этап, сведение матрицы к тридиагональной:

Алгоритм:

Все элементы матрицы, нарушающие условие тридиагональности, т. е. $A_{1,3}, A_{1,4}, \ldots, A_{1,N}, A_{2,4}, A_{2,5}, \ldots, A_{2,N}, \ldots, A_{N-2,N}$, зануляются в указанном порядке.

Зануление элемента A_{ik} достигается элементарным вращением в плоскости $(i+1, k)$, т. е. внедиагональный элемент унитарной матрицы U лежит на одну строку ниже, чем текущий элемент A_{ik}. Соответствующее элементарное вращение определяется условием:

$$A_{i,i+1}U_{i+1,k} + A_{i,k}U_{k,k} = 0.$$

имеющим вид $(*)$.

Ни одно из вращений не портит результата, достигнутого всеми предыдущими вращениями, т. е. все, обращенное ранее в нуль, нулем и остается. Поэтому нужный результат достигается за один проход.

Здесь намеренно приведен не самый эффективный, но зато самый «понятный» и простой в написании алгоритм.

2-й этап, диагонализация тридиагональной матрицы:

На этом этапе необходимо позабыть о том, что мы имеем дело с матрицей. Мы располагаем диагональными элементами и элементами, скажем, нижней субдиагонали. Надежнее всего так и хранить их в двух одномерных массивах. (Элементы верхней субдиагонали им комплексно сопряжены.) На этом 2-м этапе нам, разумеется, понадобится функция, реализующая элементарное вращение. Есть очень большой соблазн воспользоваться функцией, которая использовалась на 1-м этапе. Так вот, печальный опыт показывает, что делать это ни в коем случае нельзя. Дело опять-таки не только в экономии времени. Дело в том, что ошибки округления, сидящие в элементах $A_{i+2,\,i}$ (эти элементы в настоящей тридиагональной матрице равны нулю), могут фатально сказаться на точности алгоритма. Для реализации элементарного вращения следует выписать явные формулы для преобразования элементов диагонали, субдиагонали и одного (точнее говоря, двух, но в данном алгоритме — одного) ненулевого элемента субсубдиагонали, появляющегося при вращении. Прочие элементы матрицы следует полагать тождественно равными нулю.

Алгоритм:

Для всех элементов нижней субдиагонали $A_{i+1,i}$, начиная с $A_{2,1}$ и до $A_{N,\,N-1}$, проделывается следующее:

Проверяется абсолютная величина $A_{i+1,i}$. Если $|A_{i+1,i}| < \epsilon$, то переходят к следующему элементу $A_{i+2,i+1}$, при этом можно считать, что матрица начинается с индекса $i+1$.

Для всех $s = i+1, \ldots, N-1$ проверяется абсолютная величина $A_{s+1,s}$. Если найдется $|A_{s+1,s}| < \epsilon$, то текущим эффективным концом матрицы N' считают $N' = s$. Если такого элемента не найдется, то $N' = N$.

Вычисляются λ_1, λ_2 — СЗ матрицы 2×2:

$$\begin{pmatrix} A_{i,i} & A_{i,i+1} \\ A_{i+1,i} & A_{i+1,i+1} \end{pmatrix}$$

$$\lambda_{1,2} = \frac{1}{2}\left[A_{i,i} + A_{i+1,i+1} \pm \sqrt{(A_{i,i} - A_{i+1,i+1})^2 + 4|A_{i,i+1}|^2}\right]$$

— и из них выбирается один λ — ближайший к $A_{i,i}$.

Совершают элементарное вращение в плоскости $(N'-1, N')$, которое определяется условием

$$A_{N',N'-1}U_{N'-1,N'-1} + A_{N',N'}U_{N',N'-1} = 0.$$

Результатом этого вращения будет, во-первых, изменение двух диагональных элементов ($A_{N',N'}$ и $A_{N'-1,N'-1}$) и субдиагонального элемента $A_{N',N'-1}$. Во-вторых, появится один субсубдиагональный элемент

$$A_{N',N'-2} = U^{+}{}_{N',N'-1}A_{N'-1,N'-2}.$$

(Разумеется, заводить еще один одномерный массив для этого элемента не следует — хватит и одной переменной.)

Цепочкой элементарных вращений в плоскостях $(N'-2, N'-1), (N'-3, N'-2), \ldots (i, i+1)$ восстанавливают трехдиагональность матрицы:

Каждое из этих вращений в плоскости $(r, r+1)$ определяется условием

$$A_{r+2,r}U_{r,r} + A_{r+2,r+1}U_{r+1,r} = 0$$

и тем самым зануляет субсубдиагональный элемент $A_{r+2,r}$. Одновременно оно порождает субсубдиагональный элемент $A_{r+1,r-1} = U^{+}_{r+1,r}A_{r,r-1}$, который будет зануляться уже следующим вращением (в плоскости $(r-1, r)$).

На последнем этапе (вращение в плоскости $(i, i+1)$) трехдиагональность будет окончательно восстановлена. Дело в том, что раз мы дошли до элемента $A_{i+1,i}$, то $A_{i,i-1}$ равен нулю (с точностью до ϵ), т. е. матрица эффективно начинается с индекса i. Поэтому никакого $A_{i+1,i-1} = U^{+}_{i+1,i}A_{i,i-1}$ не появится.

Как ни странно, результатом этой (на первый взгляд, довольно бессмысленной) процедуры будет существенное уменьшение субдиагонального элемента $A_{i+1,i}$.

Как правило, для обращения $A_{i+1,i}$ в (почти) нуль требуется не более десятка итераций.

Для сходимости абсолютно необходимо проводить поиск N' на каждом обороте процедуры. В противном случае, наткнувшись по дороге на

элемент $|A_{s+1,s}| < \epsilon$, процедура будет все последующие повороты проводить на углы порядка ϵ, и текущий элемент субдиагонали $A_{i+1,i}$ будет также преобразовываться с помощью поворота на угол порядка ϵ, т. е. никакой сходимости не будет.

Поясним теперь название алгоритма. Собственно, именно второй этап алгоритма и называется LQ-алгоритмом (впрочем, в данном случае его следует называть QR-алгоритмом). Выясним, какими свойствами обладает унитарная матрица U_I, равная произведению всех элементарных вращений, производимых за одну итерацию. Умножение справа на матрицу первого элементарного вращения в плоскости $(N'-1, N')$ обращает в ноль элемент $A_{N',N'-1}$ по построению. Легко проверить, что умножение справа на матрицы всех последующих элементарных вращений (в плоскостях $(N'-2,\ N'-1), (N'-3,\ N'-2), \ldots, (i,\ i+1)$) обращает в ноль все остальные элементы нижней субдиагонали. Действительно, умножение справа на матрицу элементарного вращения в плоскости $(r,\ r+1)$ перемешивает r-й и $r+1$-й столбцы. При этом из всех поддиагональных элементов поменяться могут только элементы нижней субдиагонали, т. е. все остальные поддиагональные элементы остаются равными нулю. Аналогично, при последующем умножении слева на эрмитово-сопряженную матрицу элементарного вращения в плоскости $(r,\ r+1)$ перемешиваются r-я и $r+1$-я строки. Следовательно, если хотя бы один элемент нижней субдиагонали $R_{r-1,r}$ матрицы $R = AU_I$ был бы отличен от нуля, то после этого умножения с неизбежностью появился бы ненулевой элемент субсубдиагонали $A_{r-1,r+1}$, который нарушал бы условие тридиагональности. Следовательно, матрица $R = AU_I$ является верхней (правой) треугольной. Термин QR-алгоритм возник потому, что матрицу $U^+{}_I$ обыкновенно обозначают Q, так что за одну итерацию мы совершаем унитарное вращение $A \to QAQ^+ = QR$, притом, что на матрицу $R = AQ^+$ накладывается условие верхней (правой=Right) треугольности. Аналогично, если бы мы работали с верхней субдиагональю, то $A \to Q^+AQ = LQ$, причем на $L = Q^+A$ накладывается условие нижней (левой=Left) треугольности, то мы получили бы LQ алгоритм — зеркальное отражение QR алгоритма.

Осталось пояснить, отчего эта процедура вообще сходится. Проще всего понять это на примере матрицы 2×2

$$A = \begin{pmatrix} a & b^* \\ b & c \end{pmatrix}.$$

Пусть b не слишком велико, а a и c не слишком близки друг к другу. Тогда $\lambda_1 = a + \delta$, где δ невелико. (В данном случае $\delta = O(|b|^2/(c-a))$,

но в общем случае оно зависит и от других субдиагональных элементов.) При этом $\lambda_2 \approx c$, так что СЗ λ, ближайшее к a, есть λ_1. Поэтому первое элементарное вращение

$$U = \begin{pmatrix} \alpha & -\beta^* \\ \beta & \alpha^* \end{pmatrix}$$

определяется условием

$$b\alpha + (c-a)\beta = 0.$$

После первого (и в данном случае единственного) элементарного поворота субдиагональный элемент станет равен

$$\begin{aligned} b \to -\beta(\alpha\delta + \beta b^*) = \\ = \frac{b}{\sqrt{|c-a|^2+|b|^2}}\left(\frac{c-a}{\sqrt{|c-a|^2+|b|^2}}\delta - \frac{b}{\sqrt{|c-a|^2+|b|^2}}b^*\right) = \\ = \frac{b[\delta(c-a) - |b|^2]}{|c-a|^2+|b|^2} \approx b\frac{\delta}{c-a}. \end{aligned}$$

Итак, если спектр матрицы не является (почти) вырожденным (т. е. если $c-a$ не слишком мало), то субдиагональный элемент действительно существенно уменьшается даже за одну итерацию. Как показывает опыт, LQ-алгоритм позволяет диагонализовывать даже матрицы со спектром, который довольно сильно вырожден. При этом в окончательной диагонализованной матрице одинаковые собственные значения просто не идут подряд, т. е. не стоят рядом на диагонали.

Разумеется, приведенное рассуждение является только пояснением, а никак не доказательством сходимости LQ-алгоритма в общем случае.

14.3. Неэрмитова матрица

Совершенно ясно, что для произвольной неэрмитовой матрицы можно применить как метод Якоби, так и LQ-алгоритм, но при этом мы сможем обратить в нуль (с точностью до ϵ) либо поддиагональные, либо наддиагональные элементы, т. е. в результате унитарного преобразования матрица станет верхней (правой) или нижней (левой) треугольной. Это почти завершает задачу поиска ее СЗ и СВ. Действительно, пусть матрица стала правой треугольной: $U^+AU = R$, т. е. $AU = UR$. Тем самым один правый СВ $\vec{x}^{(1)}$

и соответствующее СЗ λ_1 мы знаем — $x_i^{(1)} = U_{i1}$, $\lambda_1 = R_{11}$. Второй правый СВ $\vec{x}^{\,(2)}$ (который, разумеется, отнюдь не обязан быть ортогональным первому СВ $\vec{x}^{\,(1)}$) строится как линейная комбинация первого и второго столбцов матрицы U. Действительно, полагая $x_i^{(2)} = U_{i2} + \alpha_1 U_{i1}$, получим

$$\lambda_2(U_{i2} + \alpha_1 U_{i1}) = R_{22}U_{i2} + R_{12}U_{i1} + \alpha_1 R_{11}U_{i1},$$

откуда, приравнивая коэффициенты при U_{i1} и U_{i2}, имеем

$$\lambda_2 = R_{22} \qquad \alpha_1 = R_{12}/(R_{22} - R_{11}).$$

Наличие возможного нулевого знаменателя в данном случае нас тревожить не должно, т. к. нулевой знаменатель означает попросту вырождение пространства собственных векторов (как, например, у матрицы $\left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$).

Для s-го СВ положим $x_i^{(s)} = U_{is} + \sum_{k=1}^{s-1} \alpha_k U_{ik}$, тогда получим

$$\lambda_s(U_{is} + \sum_{k=1}^{s-1} \alpha_k U_{ik}) = R_{ss}U_{is} + \sum_{n=1}^{s-1} R_{ns}U_{in} \sum_{k=1}^{s-1} \alpha_k \sum_{n=1}^{k} R_{nk}U_{in}.$$

Здесь надо приравнять коэффициенты при $U_{i1}, U_{i2}, \ldots, U_{is}$ в левой и правой частях равенства. Тогда получим, во-первых, $\lambda_s = R_{ss}$. Во-вторых, система линейных уравнений (СЛУ) для α_k оказывается треугольной:

$$R_{ss}\alpha_{s-1} = R_{s-1\,s} + \alpha_{s-1}R_{s-1\,s-1}$$
$$R_{ss}\alpha_{s-2} = R_{s-2\,s} + \alpha_{s-1}R_{s-2\,s-1} + \alpha_{s-2}R_{s-2\,s-2},$$

и т. д. Заметим, что при решении этой СЛУ обратной подстановкой каждый раз будут возникать знаменатели вида $1/(R_{ss} - R_{kk})$.

Итак, если нам нужны только СЗ, то никаких дополнительных манипуляций не требуется, а если нужны еще и правые СВ, то каждый из них находится посредством решения СЛУ с треугольной матрицей, т. е. решение идет просто обратной подстановкой.

Довольно ясно, что если нам нужны левые, а не правые СВ, то зануллять следует не поддиагональные, а наддиагональные элементы, т. е. должно получиться $U^+AU = L$, где L — левая (нижняя) треугольная матрица.

14.4. Вариационный метод

Довольно часто нам необходимо найти не все, а только некоторые СЗ и СВ матрицы. Типичный пример — надо найти несколько нижних уровней гамильтониана H.

В этом случае самое простое — применить вариационный метод Ритца, известный нам из квантовой механики.

Условный экстремум «функционала» (в данном случае просто функции N аргументов) $F(\vec{x}) = \vec{x}\hat{H}\vec{x}$ при условии $\vec{x}\cdot\vec{x} = 1$ даст волновую функцию $\vec{x}_0$ — основное состояние системы с гамильтонианом $\hat{H}$. Его энергия равна $E_0 = \vec{x}_0\hat{H}\vec{x}_0$.

Если мы нашли основное состояние, то легко найти и первый возбужденный уровень E_1. Для этого достаточно добавить условие ортогональности $\vec{x}_0$, т. е. $\vec{x}\cdot\vec{x}_0 = 0$. Чтобы найти второй возбужденный уровень, надо добавить условие $\vec{x}\cdot\vec{x}_1 = 0$, и так далее.

Формально говоря, чтобы найти основное состояние, мы должны искать безусловный экстремум функции $N+1$ аргумента ($\vec{x}$ и λ):

$$G(\vec{x},\lambda) = \vec{x}\hat{H}\vec{x} - \lambda(\vec{x}\cdot\vec{x} - 1);$$

здесь λ — множитель Лагранжа, играющий роль энергии в уравнении Шредингера. Действительно, условия экстремума имеют вид:

$$\hat{H}\vec{x} - \lambda\vec{x} = 0 \qquad \vec{x}\cdot\vec{x} - 1 = 0.$$

Разумнее, однако, не работать с функцией $G(\vec{x},\lambda$ непосредственно, ведь это увеличивает количество переменных в задаче. Исключая λ, получим

$$\hat{H}\vec{x} - \frac{\vec{x}\hat{H}\vec{x}}{\vec{x}\cdot\vec{x}}\vec{x} = 0.$$

Это уравнение соответствует безусловному минимуму функции

$$G_0(\vec{x}) = \frac{\vec{x}\hat{H}\vec{x}}{\vec{x}\cdot\vec{x}}.$$

Условие нормировки $\vec{x}\cdot\vec{x} = 1$ при этом накладывать необязательно.

Итак, чтобы найти основное состояние системы, надо найти минимум функции $G_0(\vec{x})$. Следует иметь в виду, что если минимум $G_0(\vec{x})$ ищется динамическим методом, то целесообразно на каждой итерации нормировать вектор $\vec{x}$ (т. е. поделить каждую компоненту $\vec{x}$ на $\sqrt{\vec{x}\cdot\vec{x}}$) — это ускоряет сходимость.

Чтобы найти первое возбужденное состояние, надо искать минимум функции

$$G_1(\vec{x}) = \frac{\vec{x}\hat{H}\vec{x} + C_0\cdot(\vec{x}\cdot\vec{x}_0)\cdot(\vec{x}_0\cdot\vec{x})}{\vec{x}\cdot\vec{x}}.$$

Здесь $\vec{x}_0$ — волновая функция основного состояния, а C_0 — произвольная, но достаточно большая константа. Формально говоря, C_0 должно удовлетворять условию $E_0 + C_0 > E_1$. Значение константы C_0 влияет главным образом на точность определения волновой функции, на точность определения энергии она почти не влияет. Точность определения волновой функции можно контролировать по произведению $\vec{x} \cdot \vec{x}_0$, которое должно равняться нулю.

Аналогично, чтобы найти второе возбужденное состояние, надо искать минимум функции

$$G_2(\vec{x}) = \frac{\vec{x}\hat{H}\vec{x} + C_0 \cdot (\vec{x} \cdot \vec{x}_0) \cdot (\vec{x}_0 \cdot \vec{x}) + C_1 \cdot (\vec{x} \cdot \vec{x}_1) \cdot (\vec{x}_1 \cdot \vec{x})}{\vec{x} \cdot \vec{x}}.$$

Здесь $\vec{x}_0$ и $\vec{x}_1$ — волновые функции основного и первого возбужденного состояний, а C_0 и C_1 — произвольные (но достаточно большие) константы.

И так далее.

Разумеется, этим способом следует пользоваться только в том случае, когда необходимо найти лишь несколько нижних уровней. Зато он пригоден и для того случая, когда LQ-алгоритм не сходится за конечное время, и, более того, для того случая, когда матрица H вообще не помещается целиком в памяти машины, но зато мы в состоянии вычислить $\vec{x}\hat{H}\vec{x}$.

15. Задачи для вычислительного практикума

Предлагаемый набор задач (точнее, описаний того, как можно соорудить задачи) ориентирован на лекционный материал, изложенный в пособии. При этом предполагается, что студенты умеют программировать на языке «C». Ввиду личных склонностей авторов речь идет именно о «C», а не о «C++». Поскольку реальная научная работа может вестись только в среде Unix (речь идет о любом варианте — Linux, Solaris, BSD, ...), то при обучении никаких оболочек не предоставляется, а предлагается пускать компилятор/линкер Watcom 10.0 (или 11.0)[9] из командной строки, не пользоваться вводом с клавиатуры, а вывод на экран

[9]По мнению авторов, на настоящий момент это лучшая реализация языка «C» в мире DOS/Windows. В нем не так много ошибок и достаточно хороший оптимизатор. При правильной оптимизации программа считает гораздо быстрее, чем при использовании других компиляторов. Кроме того, DOS extender dos4gw, под который компилирует Watcom, гораздо аккуратнее и качественнее, чем другие. В частности, он (как это и должно быть) детектирует обращение к чужой области памяти. Так

использовать только для целей трассировки. Входные данные программа читает из файла, выдачу пишет тоже в файл. Тем самым создаются условия, максимально напоминающие работу в среде Unix с удаленного терминала.

Опыт показывает, что за семестр можно выполнить около 5 задач из предлагаемого перечня. При выдаче и проверке заданий использовалась такая технология: в открытый доступ выкладывается текст всех вариантов задачи, входные файлы с параметрами (если они есть) и программа "`check`", которая проверяет (с учетом варианта) выходной файл, созданный при работе программы, написанной студентом. Преподаватель же просматривает студенческую программу, или когда она прошла проверку с помощью "`check`", или когда студент заходит в тупик (что является более типичной ситуацией). Заметим, что `check` может сравнивать ответ студента либо с тем ответом, который сосчитал преподаватель, либо с точным ответом (если он известен). В последнем случае преподавателю все равно совершенно необходимо сосчитать задачу — просто чтобы знать, какую точность требовать со студентов. При высоком уровне студентов в задачах необязательно указывать метод решения. При среднем уровне это совершенно необходимо.

Текущий набор задач (условия, входные файлы и программы "`check`"), а также некоторые старые задачи выложены в Интернет. Попасть туда можно или со странички кафедры квантовой теории и физики высоких энергий физического факультета МГУ:

`http://hep.itpm.msu.su`,

или непосредственно:

`http://tex.bog.msu.ru/numtask`.

Задача 1:

Построить псевдослучайную последовательность `unsigned long` чисел $a_1 \ldots a_{1\,000\,001}$ по следующему алгоритму:

$$a_{i+1} = a_i * 1664525 + 1013904223; \qquad a_0 = 111 * N$$

(здесь N — номер варианта). Упорядочить эти 1 000 001 чисел в порядке возрастания методами `qsort` и `heapsort`. Сравнить их ско-

что перенос программ с Watcom на компилятор GNU C (gcc под Unix) идет гораздо легче, чем с Microsoft или Borland.

рости. Выходной файл должен содержать следующие элементы упорядоченного массива:

$$a_1, a_{10\,001}, a_{20\,001}, a_{30\,001} \ldots a_{1\,000\,001}$$

(101 элемент) в указанном порядке.

Эту задачу удобно давать в самом начале семестра. В некотором смысле она является «разминочной» — по результатам ее выполнения можно судить о возможностях студентов. Другого (особого) смысла у этой задачи нет. Если уровень студентов не очень высок, то можно ограничиться задачей с одним из методов — или `qsort`, или `heapsort`.

Задача 2:

Вычислить с точностью до 40 верных знаков выражение (это один из вариантов):

$$\sum_{i=0}^{3000} \frac{\prod_{k=0}^{i-1}(k+0.1)}{\prod_{k=0}^{i-1}(k+0.2)} \Big(-0.965\Big)^i$$

(здесь $\prod_{k=0}^{i-1}(k+0.1) = 1$). Выходной файл должен содержать ответ в формате, аналогичном `%1f`, т. е.

```
0.12345678901234567890123456789012345,
```

или

```
-0.12345678901234567890123456789012345,
```

или

```
1.23456789012345678901234567890123456.
```

Разумеется, вычисляемое здесь выражение можно варьировать как угодно.

Опять-таки, это чисто тренировочная задача. Трудновато дать осмысленную задачу, которая действительно требует арифметики повышенной точности. (Например, пришлось бы долго пояснять, что такое температурная регуляризация в модели гибридных киральных мешков.) Впрочем, эта задача позволяет отбить охоту пользоваться при решении таких задач системами типа Matlab, Maple или Mathematica. Меняя верхний предел суммирования в зависимости от скорости машины, легко добиться, чтобы на «С» эта задача считалась несколько минут, а на упомянутых системах — несколько часов.

Задача 3:

Во входном файле заданы значения некоторой функции в 30 точках. Формат входного файла:

$x_0\ y_0$
$x_1\ y_1$
...
$x_{29}\ y_{29}$.

Выполнить интерполяцию функции методом кубического сплайна в точки $\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{99}$, где $\tilde{x}_i = -9.0+0.18*i$. Выходной файл должен содержать значения функции $\tilde{y}_0, \tilde{y}_1, \ldots, \tilde{y}_{99}$ в указанном порядке.

Собственно сплайновая интерполяция вещь не очень-то полезная, но эта задача вынуждает решать тридиагональную систему, что необходимо уметь делать.

Задача 4:

Значения действительной функции $f(x)$ заданы в некоторых точках действительной оси (см. входной файл). Реализовать аналитическое продолжение функции в комплексную плоскость. Выходной файл должен содержать мнимую часть функции в точках на прямой

$$\mathrm{Im}\ x{=}0.5{*}i, \qquad \mathrm{Re}\ x{=}-3.0,\ -2.98,\ -2.96, \ldots, -3.0,$$

т. е.

$$\mathrm{Im}\ f(-3.0{+}0.5{*}i) \quad \mathrm{Im}\ f(-2.98{+}0.5{*}i) \quad \ldots \quad \mathrm{Im}\ f(3.0{+}0.5{*}i).$$

а) Использовать полиномиальную интерполяцию.
б) Использовать рациональную интерполяцию.

Это более-менее осмысленная задача. Она позволяет, во-первых, заставить реализовать комплексную арифметику (и убедиться, что это очень просто), во-вторых, продемонстрировать, что «численное» аналитическое продолжение функций в комплексную плоскость возможно, в-третьих, вынуждает проверять погрешность интерполяции, в-четвертых, показывает разницу между полиномиальной и рациональной интерполяцией. Разумеется, здесь следует брать функцию, которая имеет полюса в окрестности

рассматриваемых точек. Например, можно взять линейную комбинацию $\tanh(x)$ и $1/\cosh(x)$, что-нибудь вроде

$$1.5\tanh(2x) + 0.5/\cosh(2x).$$

Возможности варьирования задачи также очень большие: можно просить вычислить действительную, а не мнимую часть функции; можно отказаться от пункта про полиномиальную интерполяцию; при невысоком уровне студентов можно отказаться от выхода в комплексную плоскость и требовать простой интерполяции на действительной оси.

Задача 5:

Методом Ньютона найти корни системы 50-и уравнений для 50-и неизвестных $\vec{f}(\vec{x}) = 0$ ($\vec{x} = (x_0, \ldots x_{49})$ и $\vec{f}(\vec{x}) = (f_0(\vec{x}), \ldots f_{49}(\vec{x}))$), где

$$\begin{aligned} f_i(\vec{x}) = {} & 0.1 * \sinh(x_i) + (x_i - x_{i-1})^3 + (x_i - x_{i+1})^3 + \\ & + 0.11 * (x_i - x_{i-13}) + 0.11 * (x_i - x_{i+13}) + \\ & + \delta_{i,0} * (x_i - 11.0) + \delta_{i,21} * (x_i + 11.0), \end{aligned}$$

при этом на индекс наложено условие цикличности, т. е. x_i при $i < 0$ означает x_{50+i}, а x_i при $i > 49$ означает x_{i-50}.

Это чисто тренировочная задача без особого физического смысла. Разумеется, система уравнений в этой задаче есть просто условие равновесия решеточной системы с простеньким нелинейным гамильтонианом. Варьировать здесь можно константы (в разумных пределах).

Задача 6:

Вычислить интеграл

$$\begin{aligned} & \int_0^1 dx \int_0^1 dy \int_0^1 dz \Big[\exp\big((x*y + y*z + z*x)/2\big) - 1.0 \Big] * \\ & * \Big[\exp\big((x^2*y + y^2*z + z^2*x + x*y*z)/2\big) - 1.0 \Big] * \\ & * \log\Big[1 + (x+y+z) + \log\big(1 + (x^2+y^2+z^2)\big)\Big] * \\ & * \log\Big[1 + 131*x^2*y^2 + 121*x^2*y*z + \\ & \qquad + 111*y^3*z + 121*y^2*z^2 + 131*y*z^3\Big] \end{aligned}$$

с относительной погрешностью не более 10^{-12}.

Это вполне осмысленная задача (несмотря на явную бессмысленность подынтегральной функции). Во-первых, задача заставляет вычислить не просто одномерный интеграл, а интеграл повторный. Это позволяет напомнить, что в языке «С» бывают указатели на функции — студенты, как правило, этого не помнят. Во-вторых, она успешно отбивает желание использовать метод Симпсона (что всегда является первым порывом студента, получившего эту задачу). Одномерный интеграл методом Симпсона взять еще можно, а здесь отношение скоростей возводится в куб, так что 5 минут (время указано для Pentium 100) заменится на сутки.

К сожалению, варьировать здесь можно только саму подынтегральную функцию. При этом не следует слишком увлекаться — она должна оставаться в значительной мере полиномиальной (как в приведенном примере), иначе интеграл будет считаться дольше.

Задача 7:

Найти эффективный потенциал в одномерной решеточной модели типа Гросса–Невье с динамическим нарушением симметрии. Эффективный потенциал есть сумма упругой энергии бозонных степеней свободы x_i, $i = 0, \ldots, N-1$:

$$V_1(x) = 0.5 \sum_{i=0}^{N-2} (x_{i+1} - x_i)^2,$$

и перенормированной энергии Казимира фермионных степеней свободы

$$V_2(x) = \sum_{i=0}^{N/2-1} (\epsilon_i(x) - \epsilon_i(x=0)).$$

Здесь предполагается, что энергии упорядочены ($\epsilon_i(x) \leqslant \leqslant \epsilon_{i+1}(x)$), т. е. сумма идет по нижней половине спектра. Уровни энергии фермионов $\epsilon_i(x)$ суть собственные значения матрицы

$$\begin{aligned} H[i][j] = {} & \delta_{i+1,j}[10.724910 + 5.149937 * (x_j - x_i)] + \\ & + \delta_{i,j+1}[10.724910 + 5.149937 * (x_i - x_j)] + \\ & + \delta_{i+5,j}[1.014994 + 0.417762 * (x_j - x_i)] + \\ & + \delta_{i,j+5}[1.014994 + 0.417762 * (x_i - x_j)]. \end{aligned}$$

Найти эффективный потенциал $V(x) = V_1(x) + V_2(x)$ для конфигурации

$$x_i = \phi(-1)^i, \qquad i = 0, \ldots N-1$$

при $N = 444$, $\phi = 0.01, 0.02, 0.03, \ldots 0.4$. Точность диагонализации (точность зануления внедиагональных элементов) — не менее 10^{-8}. Формат выходного файла — 40 чисел (значений $V(\phi)$) в порядке возрастания ϕ. Обратите внимание, что минимум достигается при $\phi \neq 0$ — это и есть динамическое нарушение симметрии. Время работы программы должно составлять около 15 минут.

Это вполне осмысленная задача («из жизни физика-теоретика»), но давать ее можно только при достаточно высоком уровне студентов. Варьировать здесь можно константы (в разумных пределах) и расстояние «дальнодействия».

Задача 8:

Найти профиль скирмиона со спином. Для этого следует найти минимум эффективного потенциала модели на решетке:

$$V(\varphi_0, \varphi_1, \ldots \varphi_N) = \Big[\alpha \sum_{i=0}^{N} \sin^2(\varphi_i)\, dr + \\ + \beta \sum_{i=0}^{N} r_i^{-2} \sin^4(\varphi_i)\, dr + \gamma \sum_{i=0}^{N-1} r_i^2 (\varphi_{i+1} - \varphi_i)^2/dr + \\ + \delta \sum_{i=0}^{N-1} \sin^2(\varphi_i)(\varphi_{i+1} - \varphi_i)^2/dr\Big] \times \\ \times \Big[1 + S(S+1) \sum_{i=0}^{N} \sin^4(\varphi_i)\, dr\Big]^{-1},$$

где α=1.4847620, β=1.3631090, γ=1.9324320, δ=1.9879150, $S = 1$. При этом $N = 400$, а $r_i = 1.0 \cdot 10^{-2} + 0.025 * i$. Кроме того, наложены следующие граничные условия: $\varphi_0 = = \pi$, $\varphi_N = 0$. Кроме того, зависимость φ_i от индекса i долж-

на быть гладкой. Минимум должен быть найден с точностью до 10^{-10} по градиенту, т. е. абсолютная величина градиента функции $V(\varphi_0, \varphi_1, \ldots, \varphi_N)$ по каждой из переменных φ_1, $\varphi_2, \ldots, \varphi_{N-1}$ не должна превышать 10^{-10}. Выходной файл должен содержать значения $\varphi_0, \varphi_1, \ldots, \varphi_N$.

Это тоже вполне осмысленная задача («из жизни физика-теоретика») и притом довольно простая. Она вынуждает использовать динамический метод (метод сопряженных градиентов не даст нужную точность). Кроме того, задача вынуждает проверять полученный ответ — если взять негладкое начальное приближение (вместо «естественной» линейной функции индекса, меняющейся от π до нуля, студенты почему-то берут тождественный нуль для $\varphi_1 \ldots \varphi_{399}$), то и ответ будет негладкий. Варьировать здесь можно константы. Если хочется упростить задачу, то можно уничтожить спин (знаменатель выражения).

Задача 9:

На плоскости (x, y) расположена прямоугольная решетка с узлами в точках $\vec{x}_{ij}=(i * dx, j * dy, 0)$, где $i, j=0 \ldots 24$, $dx = dy = 1.0$. В узлах решетки находятся точечные заряды q_{ij}. На высоте $h = 5.007241$ над плоскостью в точках $\vec{y}_{ij} = (i * dx, j * dy, h)$ измеряется компонента электрического поля, перпендикулярная плоскости: $w_{ij}^{(in)} \equiv E_z(\vec{y}_{ij})$. Входной файл содержит значения $w_{ij}^{(in)}$ в следующем порядке:

$$w_{0,0}^{(in)},\ w_{0,1}^{(in)},\ w_{0,2}^{(in)},\ \ldots\ w_{0,24}^{(in)},\ w_{1,0}^{(in)},\ w_{1,1}^{(in)},\ w_{1,2}^{(in)},\ \ldots$$

Найдите заряды $q_{ij}^{(N)}$, соответствующие полям $w_{ij}^{(N)} = w_{ij}^{(in)} + + N * 0.0005 - 0.1$, где $N = 0, 1, \ldots, 400$. Оцените гладкость распределения зарядов $q_{ij}^{(N)}$ по формуле

$$S^{(N)} = \sum_{ij} (-1)^{i+j} q_{ij}^{(N)}.$$

Убедитесь, что малая гладкая погрешность («сдвиг нуля у прибора») в определении поля $\delta w_{ij}^{(N)} = N * 0.0005 - 0.1$

приводит к существенно негладким распределениям зарядов. Выходной файл должен содержать значения $S^{(N)}$ для $N = 0 \ldots 400$. Программа должна содержать проверку обратной подстановкой (по найденным зарядам найти поля и сравнить с исходными, погрешность должна быть порядка 10^{-15}). Время счета программы должно составлять около 3 минут.

Это тоже задача «из жизни физика-теоретика». Ее чудовищная формулировка проистекает из опыта многолетней борьбы со студентами. Само по себе решение системы линейных уравнений не представляет особого интереса. К сожалению, в данном случае как использование LU-разложения (разумный подход), так и тривиальная триангуляция (не очень разумный подход) позволяют получить ответ за конечное время. Зато эта задача представляет собой довольно наглядный пример некорректно поставленной задачи. При решении очень хорошо видно, что некорректность не имеет отношения к ошибкам при формальном решении системы линейных уравнений.

В этой задаче можно варьировать параметры; кроме того, можно давать аналогичную задачу на магнитное поле и поверхностные токи.

Задача 10:

Решить систему уравнений

$$\sum_{i=0}^{16383} A_{m-i} x_i = b_m,$$

где $m = 0, \ldots, 16383$. Значения A_i для неотрицательных индексов приведены во входном файле. Для отрицательных индексов $A_i \equiv A_{i+16384}$, $(i < 0)$. Значения b_i приведены в том же входном файле. Формат выходного файла — значения x_i. Проведите проверку обратной подстановкой, погрешность должна быть около 10^{-11}. Время счета должно составлять около 6 секунд.

Это очень простая задача на алгоритм FFT. Довольно ясно, что другим способом задачу за конечное время решить нельзя. В качестве A_i

имеет смысл брать некую гладкую функцию с пологим максимумом типа гауссовой шляпки (что-то вроде приборной функции с плохим разрешением). Возможности варьирования этой задачи довольно ограничены.

Литература

Здесь перечислены только достаточно универсальные пособия (посвященные или численным методам «вообще», или достаточно большому разделу численных методов). Обширную литературу, посвященную отдельным узкоспециальным вопросам, имеет смысл изучать тогда, когда в Вашей конкретной задаче простейшие методы не работают или оказываются неэффективными.

[1] Калиткин Н. Н. «Численные методы», М.: «Наука», 1978.

[2] Бахвалов Н. С., Жидков Н. П., Кобельков Г. М. «Численные методы», М.: «Наука», 1973; М.: «Наука», 1987.

[3] Самарский А. А. «Введение в численные методы», М.: «Наука», 1982; М.: «Наука», 1987; М.: «Наука», 1997.

[4] Турчак Л. И., «Основы численных методов», М.: «Наука», 1987.

[5] Крылов В. И., Бобков В. В., Монастырский П. И. «Начала теории вычислительных методов», Минск, «Наука и техника», 1986.

[6] Волков Е. А. «Численные методы», М.: «Наука», 1982.

[7] Приклонский В. И. «Численные методы в физике», М.: МГУ, 1999.

[8] Косарев В. И. «12 лекций по вычислительной математике», М.: МФТИ, 1995.

[9] Гавурин М. К. «Лекции по методам вычислений», М.: «Наука», 1971.

[10] Самарский А. А. «Введение в теорию разностных схем», М.: «Наука», 1971.

[11] Самарский А. А., Гулин А. В. «Численные методы математической физики», М.: «Научный мир», 2000.

[12] Тихонов А. Н., Арсенин В. Я. «Методы решения некорректных задач», М.: «Наука», 1986.

[13] Рихтмайер Р., Мортон К. «Разностные методы решения краевых задач», М.: «Мир», 1972.

[14] Stoer J., Bulirsh R. «Introduction to numerical analysis», New-York: Springer-Verlag, 1980.

[15] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P. «Numerical recipes in C», Cambridge University Press, 1995.

[16] Acton F. S. «Numerical methods that work», Washington: Mathematical association of America, 1990.

[17] Knuth D. E. «The art of computer programming», Reading, MA: Addison-Wesley, 1989.

[18] Golub G. H., Van Loan C. F. «Matrix computations», Baltimore: John Hopkins University Press, 1989.

Интересующие Вас книги нашего издательства можно заказать почтой или электронной почтой:

subscribe@rcd.ru

Внимание: дешевле и быстрее всего книги можно приобрести через наш Интернет-магазин:

http://shop.rcd.ru

Книги также можно приобрести:

1. Москва, ФТИАН, Нахимовский проспект, д. 36/1, к. 307, тел. 332–48–92 (почтовый адрес: Нахимовский проспект, д. 34).
2. Москва, ИМАШ, ул. Бардина, д. 4, корп. 3, к. 414, тел. 135–54–37.
3. МГУ им. Ломоносова (ГЗ, 1 этаж).
4. Магазины:

   Москва: «Дом научно-технической книги» (Ленинский пр., 40);
   «Московский дом книги» (ул. Новый Арбат, 8);
   «Библиоглобус» (м. Лубянка, ул. Мясницкая, 6);

   С.-Пб.: «С.-Пб. дом книги» (Невский пр., 28)

Ильина Вера Алексеевна
Силаев Петр Константинович

Численные методы для физиков-теоретиков

Часть 1

Дизайнер М. В. Ботя
Технический редактор А. В. Широбоков
Компьютерная верстка Д. П. Вакуленко
Корректор З. Ю. Соболева

Подписано в печать 29.12.02. Формат 60 × 84 1/16.
Усл. печ. л. 7,67. Уч. изд. л. 7,43.
Гарнитура Таймс. Бумага офсетная №1.
Печать офсетная. Заказ №95.

АНО «Институт компьютерных исследований»
426034, г. Ижевск, ул. Университетская, 1.
Лицензия на издательскую деятельность ЛУ №084 от 03.04.00.
http://rcd.ru E-mail: borisov@rcd.ru