Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

П. С. Мещеряков

# ПРИКЛАДНАЯ ИНФОРМАТИКА

Учебное пособие

Томск
«Эль Контент»
2012

УДК 004.94:51(075.8)
ББК 32.973.2я73
М 565

Рецензенты:
**Крайнов А. Ю.**, докт. физ.-мат. наук, профессор кафедры математической физики Научно-исследовательского института прикладной математики и механики Томского государственного университета;
**Касимов В. З.**, докт. физ.-мат. наук, профессор кафедры экономической математики, информатики и статистики ТУСУРа.

**Мещеряков П. С.**

М 565 Прикладная информатика : учебное пособие. / П. С. Мещеряков. — Томск : Эль Контент, 2012. — 132 с.

ISBN 978-5-4332-0060-9

В учебном пособии по дисциплине «Прикладная информатика» рассмотрены основные математические задачи, возникающие при моделировании, в том числе и электрических цепей, описаны алгоритмы их решения с использованием вычислительных мощностей компьютера.

Учебное пособие по дисциплине «Прикладная информатика» предназначено для студентов факультета дистанционного обучения ТУСУРа.

УДК 004.94:51(075.8)
ББК 32.973.2я73

ISBN 978-5-4332-0060-9

# ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

В процессе обучения языку программирования высокого уровня (Информатика, часть 2) вам предлагалось, как правило, составление достаточно простых программ. Они должны были помочь начинающему овладеть элементами языка, и при их составлении не требовалось тратить слишком много времени на обдумывание способа решения поставленной задачи. Часть примеров были исскуственно составлены для того, чтобы подчеркнуть особенность той или иной конструкции языка. На последующих этапах обучения задания естественным образом будут эволюционировать, усложняться, соответственно будут усложняться и методы их решения. Поэтому еще до обращения к языку программирования становится необходимым составление плана выполнения задания или, другими словами, разработка *алгоритма* решения.

*Можно сказать, что* ***алгоритм*** *— это набор инструкций, который описывает, как некоторое задание может быть выполнено.*

Другими словами, алгоритмы — это рецепты, описывающие некоторые классы управляющих процессов и процессов обработки данных. Их следует представлять себе как некоторые жесткие структуры, состоящие из блоков, построенных логично, надежно и целесообразно.

Собственно программирование как раз и заключается в проектировании и формулировании алгоритмов, с последующей записью на некотором алгоритмическом языке. Как правило, этот процесс очень сложен, требует учета многочисленных взаимно зависящих деталей и носит итеративный характер, т. е. имеет свойство неоднократно возвращаться к некоторым его этапам. При этом только в очень редких случаях существует единственное решение поставленной задачи. Часто решений так много, что для написания оптимальной программы требуется не только тщательный анализ известных алгоритмов, но также и рассмотрение наиболее частых случаев использования программы.

Практически всегда при проектировании технических устройств приходится иметь дело с *математической моделью* данного устройства. Апробация поведения устройства осуществляется на данной модели. Модель можно строить различными средствами облегчающими математические вычисления. Но порой различные

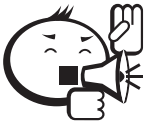
прикладные программы недоступны, в силу различных лицензионных соглашений, стоимости и т. д. Однако всегда можно написать программу практически на любом языке программирования высокого уровня, осуществляющую трудоемкие процессы математических расчетов, причем зачастую ничего не надо придумывать нового, существует множество различных алгоритмов решения маематических задач в зависимости от требуемого результата.

Рассмотрению некоторых из таких алгоритмов и посвящено данное пособие. Алгоритмы, по возможности, даны без привязки к конкретному языку программирования. Любой человек, имеющий навыки программирования, может эти алгоритмы реализовать средствами привычного ему языка программирования. В отдельных случаях, когда по мнению автора следует пояснить некоторые моменты реализации, приводятся конструкции на языке Паскаль. Данный язык выбран по причине того, что его возможностей вполне достаточно для реализации всех приведенных ниже конструкций, достаточно строгий синтаксис языка поможет миновать часть ошибок, возникающих при вычислениях, а так же имеется свободно распространяемая среда программирования FreePascal.

## Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.

*Эта пиктограмма означает определение или новое понятие.*



Эта пиктограмма означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Эта пиктограмма означает совет. В данном блоке можно указать более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.

## Контрольные вопросы по главе

# РАЗДЕЛ I

# Теоретический

# Глава 1

# ОСНОВЫ АЛГОРИТМИЗАЦИИ

## 1.1 Понятие алгоритма

Прежде чем обратиться к рассмотрению конкретных алгоритмов, необходимо определиться с тем, что же такое алгоритм. Традиционно считается, что самый первый алгоритм был придуман древнегреческим математиком Евклидом — правило нахождения наибольшего общего делителя двух целых чисел. В современной математике понятие алгоритма является ключевым понятием, которое восходит к работам выдающегося узбекского математика IX века Аль—Хорезми. В XII веке его работы по алгебре и арифметике были переведены на латынь и заложили, в значительной степени, фундамент всей европейской математики.

Примеры алгоритмов в широком смысле этого слова можно встретить и в повседневной жизни. Так, поваренная книга является сборником алгоритмов, описывающих процессы приготовления пищи. Не случайно термин «рецепт» часто можно встретить как синоним термину «алгоритм». Компьютерные программы представляют собой алгоритмы, записанные средствами языков программирования.

В качестве примера простого алгоритма рассмотрим описание процесса умножения двух целых положительных чисел $I$ и $J$, в результате которого получается произведение $P$.

***Шаг 1.*** *Присвоить P значение 0.*

***Шаг 2.*** *Присвоить K значение 0.*

***Шаг 3.*** *Если I равно 0 или J равно 0, то перейти к шагу 7.*

***Шаг 4.*** *Присвоить P значение P + I.*

***Шаг 5.*** *Присвоить K значение K* + 1.

***Шаг 6.*** *Если K меньше J, то перейти к шагу 4.*

***Шаг 7.*** *Конец.*

Таким образом, мы имеем описание некоторого задания в терминах других подзаданий, каждое из которых заранее определено либо понятно без определений.

Исчерпывающее определение ***алгоритма*** дано выдающимся отечественным математиком А. А. Марковым [5]. Алгоритм — это точное, общепринятое предписание, определяющее процесс преобразования ***исходных данных*** в ***искомый результат***. Алгоритм должен обладать следующими свойствами:

- ***определенностью,*** т. е. точностью, не оставляющей места для произвола, и общепонятностью;
- ***выполнимостью*** (или массовостью), т. е. возможностью исходить из любых исходных данных, принадлежащих некоторому множеству $G$ исходных данных;
- ***конечностью*** (или результативностью), т. е. свойством определять процесс, который для любых допустимых исходных данных (т. е. принадлежащих вышеупомянутому множеству $G$) приводит к получению искомого результата.

Рассмотрим пять признаков алгоритма более подробно. Обратившись к алгоритму умножения, отметим, что для того, чтобы он начал работу, нужно задать числовые значения $I$ и $J$ — это исходные данные (или ***вход***) алгоритма, а вычисляемая величина $P$ — это искомый результат (или ***выход***) алгоритма. В хорошо разработанном алгоритме имеются два четко различимых самостоятельных множества данных, образующих вход и выход алгоритма. Элементы этих двух множеств называются входными и выходными параметрами или аргументами.

В большинстве случаев разработка алгоритма происходит следующим образом. Начальным этапом разработки алгоритма является определение множеств входных и выходных параметров. На втором этапе формируется (возможно, на интуитивном уровне) связь между этими двумя множествами. На заключительном этапе эта связь формулируется в виде набора формальных инструкций, который и является собственно алгоритмом.

Для работоспособности алгоритма каждая из этих инструкций должна удовлетворять условиям определенности и выполнимости [4].

Для *определенности* алгоритма каждая из составляющих его инструкций должна быть определена четко и недвусмысленно. Если алгоритм выражен на какомнибудь естественном языке, то есть возможность появления неоднозначности. Рассмотрим, например, третий шаг алгоритма умножения. Если $I$ равно 0, то должен осуществится переход на шаг 7, если $J$ равно 0 — произойдет то же самое, но что произойдет, если и $I$, и $J$ равны 0? В математической логике условие «А или В» истинно, если А истинно или В истинно, или истинны и А, и В. Но в русском и, особенно, английском языках под словом «или» вполне может подразумеваться исключение случая, когда оба случая истинны (т. е. если под этим выражением подразумевается оборот «или...или...», для которого в математической логике предусмотрена операция «исключающего или» — XOR). Таким образом, пока мы оговорили точное значение термина «или», шаг 3 не будет определенным.

Для термина *«выполнимость»* ясное неформальное определение предложил Д. Кнут. Инструкция выполнима, если включенные в нее операции достаточно элементарны, чтобы их за конечное время мог выполнить человек, вооруженный

карандашом и бумагой. То, что даже простая инструкция может оказаться невыполнимой, демонстрирует следующий простой пример — вычислить наибольшее вещественное число, меньшее единицы. Ясно, что с точки зрения математики такое число определить невозможно — например, если выбрать число 0.9999, число 0.99994 будет больше и т. д.

Обеспечить выполнимость алгоритма — означает исключить из него все невыполнимые инструкции; обеспечить определенность — означает исключить неясные или бессмысленные инструкции.

В отличие от двух последних свойств *конечность* является свойством алгоритма в целом, а не отдельных его инструкций. Типичный случай алгоритма, который никогда не заканчивается — это бесконечный цикл:

***Шаг 1.*** *Присвоить I значение 0.*

***Шаг 2.*** *Присвоить I значение I* + 1.

***Шаг 3.*** *Перейти к шагу 2*.

Для любого алгоритма желательно доказать его конечность (хотя бы нефор-мально). К сожалению, такие доказательства обычно чрезвычайно сложны. При этом можно выделить два класса алгоритмов.

Во-первых, существуют итеративные численные алгоритмы, которые дают приближенные результаты. В этом случае условие завершения обычно определяется при помощи оценки значения погрешности и сравнения ее с наперед заданным критерием точности. Такие алгоритмы хорошо исследованны с математической точки зрения, но только в редких частных случаях для них имеются строгие доказательства их конечности. Сказанное не означает, что указанные алгоритмы не следует использовать (тем более часто они бывают единственно доступными способами решения), но в то же время надо сознавать возможность возникновения таких проблем.

Алгоритмы второго рода предназначены для нечисловых процессов. Для них возможно доказательство конечности и в общем виде, но часто более существенным является то, как быстро алгоритм завершается и сколько при этом требуется шагов.

Для записи алгоритмов существует несколько способов. Выше мы встретились со словесной записью алгоритма на естественном языке. Часто встречается также запись алгоритмов при помощи блок-схем и решающих таблиц. Однако для нас наиболее удобно будет записывать алгоритм либо непосредственно на языке программирования, либо на так называемом *псевдокоде*. Под псевдокодом понимают некоторый не слишком формализованный язык, промежуточный между естественным языком и языком программирования. Псевдокод позволяет программисту пользоваться достаточно произвольными словесными и математическими выражениями в сочетании с конструкциями алгоритмического языка. При этом, как правило, используются конструкции того алгоритмического языка, на котором впоследствии будет писаться программа (в нашем случае — это язык Free Pascal), хотя это и не обязательно.

## 1.2 Структурное программирование

В настоящее время большинство программистов признают, что для эффективного и быстрого проектирования надежных, «правильных» программ необходимо использовать стиль *структурного программирования*. Основные принципы, заложенные в его основу, состоят в том, чтобы при записи алгоритмов пользоваться ограниченным набором инструкций, а именно структурными операторами:

- оператором присваивания «**:=**»;
- условным оператором «**if**...**then**...**else**»;
- оператором цикла «**do**...**while**».

Хотя этих трех операторов достаточно для построения программ любой сложности, вышесказанное не надо воспринимать как догму [1]. Разумеется, вполне допустимы и другие структурные операторы — оператор выбора, другие операторы цикла и т. д. Структурное программирование предполагает при построении программ *метод пошаговой детализации*, при котором вначале разрабатывается общая структура программы, состоящая из записи алгоритма не с помощью элементарных операций, а более крупными блоками. Впоследствии, на следующих этапах каждый из этих блоков детализируется на более мелкие, пока в результате не получатся элементарные действия, которые можно выразить непосредственно средствами языка программирования.

Другими словами, *структурное программирование* — это метод программирования, обеспечивающий создание программ, структура которых ясна и непосредственно связана со структурой решаемых задач.

Из опыта производства хорошо известно, что улучшать качество готового изделия — безнадежное занятие. Несмотря на важность проверки качества, обеспечить его можно только на производственной линии. Аналогично для программ — тестирование и отладка представляют собой крайне ненадежные средства по сравнению с правильным написанием программ с самого начала. Невозможно протестировать каждый возможный случай, и поэтому его можно применять только для доказательства наличия ошибок, но не их отсутствия.

*Несколько принципов, которыми желательно пользоваться при написании программ.*

1) Не применять всевозможные трюки или заумные методы. Не надо использовать сложные методы там, где есть простые.
2) Использовать как можно меньше переходов **goto**. Автор языка Паскаль Н. Вирт предлагает их использовать только для выхода из циклов, многократно вложенных друг в друга.
3) Операторы цикла должны быть использованы самым простым образом. Не надо использовать цикл **do**...**while** там, где проще и нагляднее выглядит цикл **for**...**to**...**do**.
4) Громоздкие программы, состоящие из одной основной программы, как правило, невозможно читать. Зачастую глубина вложенности циклов и условных операторов такова, что для каждого **if**...**then** затруднительно найти соответствующий **else**. Чтобы обойти указанную трудность, каждую боль-

шую программу можно разбить на множество модулей и процедур, спроектированных так, что цель каждой из них определена (как логическая часть исходной задачи) и в каждой по возможности используются свои локальные переменные. Не надо разбивать программу на приблизительно равные «куски», старайтесь выделять логически связанные фрагменты. Даже если фрагмент программы встречается в ее тексте только один раз (например, ввод начальных данных), его разумно оформить в виде отдельной процедуры.

В заключение отметим, что хорошо построенная программа должна быть не только хорошо *структурирована*, но и хорошо *документирована*. Это нужно не только для того, чтобы Вашу программу мог читать кто-то другой (это может быть коллега или, что часто гораздо важнее, заказчик Вашей программы), но и Вы сами спустя какое-то время — например, месяц или год — после написания программы. Для этого в текст программы желательно вставлять достаточно подробные комментарии, описывающие, что и как делают отдельные операторы или группы операторов. Для имен переменных, функций и т. д. желательно использовать *мнемоничные* (т. е. содержательные) идентификаторы, по которым можно понять, для чего или как данная переменная или функция используются.

## 1.3 Пошаговая разработка программ

Как уже отмечалось выше, программирование неразрывно связано с проектированием алгоритмов. При этом часто удобно сочетать поэтапную разработку алгоритма с написанием собственно программы. Этот метод носит название *пошаговой разработки программ*.

Конструирование алгоритма требует тщательного осмысливания и исследования возможных решений. На ранних стадиях обращают внимание главным образом на глобальные проблемы, упуская из виду многие детали. По мере продвижения процесса проектирования задача разбивается на подзадачи, и постепенно все большее внимание уделяется подробному описанию проблемы и характеристикам имеющихся инструментов программирования.

Вероятно, наиболее общая тактика программирования состоит в разбиении задачи на небольшое число подзадач, каждая из которых представляет собой более простую самостоятельную задачу, а соответствующих программ на отдельные инструкции. При этом на каждом таком шаге этой декомпозиции решения частных задач должны приводить к решению общей задачи, а выбранная последовательность индивидуальных действий должна быть разумна и позволять получить инструкции более близкие к языку, на котором, в конечном счете, будет сформулированна программа.

Именно последнее требование исключает возможность прямолинейного продвижения от первоначальной постановки задачи к программе, которая должна получиться в конечном итоге. Всякий этап декомпозиции сопровождается формулированием частных программ. В процессе этой работы может обнаружиться, что выбранная декомпозиция неудачна в том или ином смысле хотя бы потому, что подпрограммы неудобно выражать с помощью имеющихся средств. В этом слу-

чае один или несколько предыдущих шагов декомпозиции следует пересмотреть заново.

Поэтапная декомпозиция, сочетающаяся с одновременной детализацией программы, называется еще методом *нисходящего проектирования* или проектированием *сверху вниз*. Возможен альтернативный подход к решению задачи, когда программист сначала изучает имеющуюся в его распоряжении вычислительную систему, а затем собирает некоторые последовательности инструкций в элементарные процедуры, типичные для решаемой задачи. Элементарные процедуры затем используются на следующем, более высоком уровне иерархии процедур. Такой метод называется *восходящим* (или *снизу вверх*). На практике разработку сколько-нибудь сложной программы никогда не удается провести строго в одном направлении (сверху вниз или снизу вверх). Однако при конструировании новых алгоритмов обычно в большей степени используется нисходящий метод. С другой стороны, при адаптации программы к несколько измененным условиям эксплуатации предпочтение зачастую отдается восходящему методу.

Если программа разбивается на подпрограммы, то для установления связи между ними часто приходится вводить новые переменные. Такие переменные следует вводить и описывать на том этапе разработки, на котором они потребовались. Более того, детализация описания процесса может сопровождаться детализацией описания структуры используемых переменных. Следовательно, в языке должны быть средства для отражения иерархической структуры данных. Из сказанного видно, какую важную роль играют такие понятия языка Паскаль, как процедуры, локальность объектов и структурирование данных в связи с пошаговой разработкой программы.

## 1.4 Рекуррентные алгоритмы

При решении различного рода математических задач часто приходится иметь дело с алгоритмами, построенными на рекуррентных соотношениях, т. е. выражениях вида

$$v_i = f(v_{i-1}) \text{ для всех } i > 0. \tag{1.1}$$

При этом последовательность $v_0, v_1, \ldots, v_n$ называется рекуррентной последовательностью. Часто используют более общее определение элементов рекуррентной последовательности при помощи формул вида

$$\begin{aligned} &v_i = f(i, v_{i-1}, v_{i-2}, \ldots, v_{i-k}),\ i \geqslant k \\ &v_0 = x_0, \\ &v_1 = x_1, \\ &\ldots, \\ &v_{k-1} = x_{k-1}. \end{aligned}$$

Однако нетрудно видеть, что на самом деле это определение становится частным случаем определения (1.1), если в качестве $v_i$ рассматривать *вектор*, одним из компонентов которого может быть собственно номер элемента рекуррентной последовательности $i$.

С самым простым примером использования рекуррентных соотношений мы сталкиваемся при определении факториала

$$f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n, \; n \geqslant 0,$$

который можно вычислить при помощи рекуррентных соотношений

$$\begin{aligned} f(0) &= 1 \\ f(i) &= i \cdot f(i-1). \end{aligned} \tag{1.2}$$

При реализации этого алгоритма на языке Free Pascal удобно заменить рекуррентные формулы (1.2) системой рекуррентных соотношений

$$\left.\begin{aligned} f_i &= k_i \cdot f_{i-1} \\ k_i &= k_{i-1} + 1 \end{aligned}\right\} \text{ для } i > 0$$

с начальными значениями

$$f_0 = 1, \quad k_0 = 0.$$

Тогда программу вычисления факториала можно записать в следующем виде

```
var f, k, n: word;
begin
  writeln ('задайте␣n');
  read (n);
  f:=1; k:=0;
  while k<n do
  begin
    k:=k+1; f:=k*f;
  end;
  writeln (f);
end.
```

Так как $k$ растет, принимая значения из последовательности натуральных чисел, а $n \geqslant 0$, то программа обязательно завершится, т. е. данный алгоритм *конечен*. Следует обратить особое внимание на то, что существенно важен порядок, в котором выполняются две выполняемые инструкции. Если их поменять местами:

$$f := k * f; \quad k := k + 1.$$

то другими станут и соответствующие им рекуррентные соотношения:

$$\begin{aligned} f_i &= k_{i-1} \cdot f_{i-1}; \\ k_i &= k_{i-1} + 1. \end{aligned}$$

В полном соответствии с идеями структурного программирования и пошаговой детализации алгоритмы такого рода можно изобразить в виде так называемой схемы программы, которая представляет собой не что иное, как вариант псевдокода:

$$\begin{aligned} &V := v_0; \\ &\textbf{while } p(V) \textbf{ do } V := f(V). \end{aligned} \tag{1.3}$$

Здесь $p$ обозначает условие (логическое выражение), $f$ — функцию, $V$ — множество всех переменных, встречающихся в программе. При работе этого алгоритма $V$ последовательно принимает значения $v_0, v_1, \ldots, v_n$, которые имеют следующие свойства:

1. $v_i = f(v_{i-1})$ для всех $i > 0$,
2. $v_i \neq v_j$ для всех $i \neq j$,
3. $\neg p(v_n)$,
4. $p(v_i)$ для всех $i < n$.

Здесь $\neg$ — математический оператор отрицания, для которого в языке Free Pascal используется оператор **not**. Таким образом, логические высказывания 1–4 истинны на каждом шаге алгоритма. При составлении конкретного алгоритма схема программы детализируется путем расшифровки и записи инструкции инициализации вектора $V$ начальным значением $v_0$, логического оператора $p(V)$ и оператора присваивания в теле цикла $V := f(V)$ при помощи операторов языка Free Pascal.

## Пример

### Вычисление квадратного корня

Пусть две последовательности $a_0, a_1, \ldots$ и $c_0, c_1, \ldots$ определяются рекуррентными соотношениями:

$$\left.\begin{aligned} a_i &= a_{i-1} \cdot \left(1 + \frac{1}{2} c_{i-1}\right) \\ c_i &= c_{i-1}^2 \cdot \frac{1}{4}\left(3 + c_{i-1}\right) \end{aligned}\right\} \text{ для } i > 0 \qquad (1.4)$$

и начальными значениями:

$$a_0 = x, \quad c_0 = 1 - x, \quad 0 < x < 2.$$

С помощью несложных математических выкладок можно показать, что:

$$a_n = \sqrt{x(1 - c_n)}.$$

Так как $|c_0| < 1$, то

$$\lim_{n\to\infty} c_n = 0, \quad \lim_{n\to\infty} a_n = \sqrt{x}. \qquad (1.5)$$

Подставляя рекуррентные соотношения (1.4) в (1.3), получим программу, которая вычисляет *приближенное значение* $\sqrt{x}$.

```
var a, c, e, x: real;
begin
  writeln (задайте x);
  read (x);
  writeln (задайте e);
```

```
    read (e);
    a:=x; c:=1-x;
    while c>e do
    begin
      a:=a*(1+0.5*c);
      c:=sqr(c)*(0.75+0.25*c);
    end;
    writeln (a);
end.
```

. . . . . .

Эта программа обязательно завершится (обладает свойством *конечности*), поскольку (1.5) гарантирует нам, что для сколь угодно малого $e$ найдется $n$, такое, что $c_n < e$.

Рекуррентные алгоритмы часто используются не только для вычисления собственно элементов рекуррентных последовательностей, но также и *рядов* чисел. Пусть задана последовательность членов ряда:

$$t_0, t_1, t_2, \ldots,$$

тогда ряд частичных сумм:

$$s_0, s_1, s_2, \ldots,$$

определяется так, что:

$$s_i = t_0 + t_1 + \ldots + t_i.$$

Если последовательность задается рекуррентным соотношением:

$$t_i = f(t_{i-1}) \text{ для } i > 0,$$

то ряд определяется формулой:

$$s_i = s_{i-1} + t_i \text{ для } i > 0,$$
$$s_0 = t_0.$$

Аналогично (1.3) приведем схему программы, позволяющую с помощью соответствующей замены $f$ и $t_0$ получать программы, которые на $i$-ом шагу будут присваивать переменной $S$ значения $s_i$.

$$\begin{aligned} &T := T_0;\ S := T \\ &\textbf{while } p(S,T) \textbf{ do begin} \\ &\quad T := f(T);\ S := S + T \\ &\textbf{end} \end{aligned} \tag{1.6}$$

Определяющие свойства схемы программы имеют вид

1. $t_i = f(t_{i-1})$ для всех $i > 0$;
2. $t_i \neq t_j$ для всех $i \neq j$;
3. $s_i = s_{i-1} + t_i$ для всех $i > 0$;
4. $\neg p(s_n, t_n)$;
5. $p(s_i, t_i)$ для всех $i < n$.

## Пример

Вычисление приближенного значения $\exp(x)$

Частичные суммы:

$$s_i = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^i}{i!} \tag{1.7}$$

можно определить рекуррентным соотношением

$$t_j = t_{j-1} \cdot \frac{x}{j},\ j > 0 \tag{1.8}$$

и начальным условием:

$$t_0 = 1.$$

Из курса математического анализа известно, что частичная сумма (1.7) имеет предел:

$$\lim_{n\to\infty} s_n = e^x.$$

Этот ряд сходится для любого вещественного $x$, что позволяет детализацией схемы (1.6) путем подстановки в нее рекуррентных соотношений (1.8) получить следующую программу:

```
var t, s, e, x: real; k: integer;
begin
  writeln (задайте x);
  read (x);
  writeln (задайте e);
  read (e);
  t:=1; s:=t; k:=0;
  while t>e do begin
    k:=k+1; t:=t*x/k; s:=s+t;
  end;
end.
```

Точность полученной приближенной величины $s$ коррелирует некоторым образом со значением $e$, но, вообще говоря, не равна ей. Погрешность вычислений может быть сколь угодно малой при достаточно большом $k$ (как это показано в соответствующих главах курса мат. анализа), но ее конкретное значение оценивается, как правило, опытным путем.

Необходимо отметить, что построение алгоритма, подобного описанному выше, должно строиться на тщательном предварительном математическом анализе поставленной задачи. В противном случае, возможно получение неверных результатов. Например, известно, что хотя

$$\lim_{n\to\infty} \frac{1}{n} = 0$$

гармонический ряд:

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \ldots$$

расходится, т. е.

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{i} = \infty.$$

Следует обратить особое внимание на то, что в приведенных выше программах вычисления $\sqrt{x}$ и $\exp(x)$ число повторений определить не просто. Эти числа зависят не только от величины *e*, определяющей точность, но и от *быстроты сходимости ряда* (которая зависит в данных случаях от величины *x*). Поэтому использование такого рода рекуррентных соотношений требует большой осторожности, даже если математический анализ гарантирует нам абсолютную сходимость, так как на практике важна быстрая сходимость.

## 1.5 Рекурсия

Элегантным и универсальным методом, заслуживающим специального рассмотрения, является рекурсия. Рекурсия первоначально была разработана математиками как средство для определения функций и широко использовалась в математической логике.

***Рекурсивные*** *определения (будь то функции или алгоритмы) характеризуются тем, что определяемый объект сам фигурирует в определении. Или, другими словами, объект называется рекурсивным, если он содержит сам себя или определен с помощью самого себя.*

Пример

Самый простой пример — рекурсивное определение факториала:

$$0! = 1; \text{ если } n > 0, \text{ то } n! = n(n-1)!$$

Мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания.

Возможность использования рекурсивных алгоритмов в языке программирования Free Pascal обеспечивается рекурсивными определениями процедур и функций. С процедурой принято связывать некоторое множество локальных объектов, т. е. переменных, констант, типов и процедур, которые определены локально в этой процедуре, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая процедура вызывается, для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества

локальных переменных, созданного при предыдущем обращении к этой же процедуре, их значения различны. Следующие правила области действия идентификаторов позволяют исключить какой-либо конфликт при использовании имен: идентификаторы всегда ссылаются на множество переменных, созданное последним вызовом процедуры. То же правило относится и к параметрам процедуры.

Рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. Но это не значит, что при наличии таких рекурсивных определений наилучшим способом решения задачи является рекурсивный алгоритм. К сожалению, очень часто рекурсия объясняется на неподходящих примерах, что приводит к распространению предубеждения против использования рекурсии в программировании, как неэффективного средства. Например, рассмотрим функцию для вычисления факториала

```
function f(i: integer): integer;
begin
  if i>0 then
    f:=i*f(i-1)
  else
    f:=1;
end.
```

Очевидно, что здесь рекурсию можно заменить простой итерацией, а именно программой из параграфа 1.4. Можно с уверенностью сказать, что следует избегать рекурсии, если имеется *очевидное* итеративное решение поставленной задачи. Но это не означает, что надо избавляться от рекурсии любой ценой. Во многих случаях она вполне приемлема.

## 1.6 Структуры данных

На первых этапах развития вычислительной техники компьютеры предназначались для проведения сложных и длительных расчетов. Однако впоследствии оказалось, что на первый план вышла их способность хранить и обрабатывать большие объемы информации. Представляя некоторую *абстракцию* какой-то части реального мира, информация, доступная компьютеру, состоит из некоторых *данных* о действительности, которые *считаются* относящимися к решаемой задаче и из *которых* предполагается получить требуемый результат.

Концепция структур данных чрезвычайно важна в информатике. Не случайно одна из самых известных книг в этой области называется «Алгоритмы + структуры данных = программы» (ее автор — Н. Вирт — является основным разработчиком языка Паскаль). Правда, с математической точки зрения это название не совсем точно, так как мы помним, что входные и выходные данные алгоритма являются его неотъемлемой частью. В конечном счете *программы* представляют собой конкретные формулировки абстрактных *алгоритмов*, основанные на конкретных представлениях и структурах *данных*. При этом данные представляют собой абстракции реальных объектов, сформулированные в терминах конкретного языка программирования.

Как правило, разработка алгоритма начинается с разработки структур входных и выходных данных. Поскольку данные являются абстракцией или упрощением действительности, в них игнорируются некоторые свойства и характеристики, не существенные при решении данной задачи. Для того, чтобы оперировать данными в компьютерной программе, для них выбирается некоторое представление, допустимое в используемом языке программирования. В зависимости от характера используемых данных и вида их представления в компьютере (точнее, в языке программирования) выбираются или разрабатываются методы решения поставленной задачи. Впрочем, часто встречается и обратное влияние используемых методов на данные, которыми они оперируют. В процессе конструирования программы представление данных постепенно уточняется вслед за уточнением алгоритма, все более подчиняясь ограничениям, накладываемым системой программирования и применяемым методом решения задачи. Вполне может оказаться, что мы не можем написать программу для решения задачи в полной постановке — просто не умеем или не хватает ресурсов компьютера. Тогда, вероятно, придется упростить задачу и соответственно модифицировать структуры входных и выходных данных.

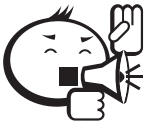
### Пример

В качестве примера рассмотрим личную карточку служащего. Каждый сотрудник может быть специфицирован в соответствующем файле при помощи некоторого набора данных, существенных либо для его характеристики, либо для процедуры расчета. В этом наборе может содержаться имя и фамилия сотрудника, его возраст, пол, заработная плата и т. д. При этом несущественные данные, такие, как цвет волос, вес и рост, вряд ли есть смысл хранить. Впрочем, наверное, в некоторых организациях вес и рост могут оказаться важны. В процессе составления программы эти данные могут быть дополнены, например порядковым номером сотрудника, используемым только как внутреннее для этой программы свойство объекта.

В заключение отметим, что требование явно описывать *любые* данные не является, по сути, требованием или ограничением собственно языка Free Pascal (в отличие, например, от языка Фортран, где переменные можно не описывать явно). Это требование (а также настоятельные рекомендации использовать там, где это возможно, вместо целых типов ограниченные типы) диктуется, по крайней мере, следующими причинами:

1) Знание диапазона значений переменных очень существенно для понимания алгоритма. От него, в большинстве случаев, зависит правомерность и корректность использования программы.
2) Как правило, многие операторы определены только для некоторых диапазонов значений.
3) Реализация многих операторов часто зависит от диапазонов принимаемых значений их аргументов.

При решении всевозможных задач обработки информации огромное значение имеют такие структуры данных, как определяемые пользователем *записи* (**record**), в то время как при решении задач численного анализа основную роль выполняют вещественные переменные и образованные из них массивы.



При построении численных алгоритмов часто приходится иметь дело с очень большими массивами вещественных чисел. Поэтому часто приходится выбирать между точностью представления вещественных чисел и объемом занимаемой ими памяти. В большинстве случаев на современных компьютерах рекомендуется пользоваться типом double, хотя в каждом конкретном случае выбор типа представления вещественных чисел лежит на программисте.

В языке Free Pascal существует изящный способ смены типа сразу у большого числа переменных. Например, если во всей программе необходимо заменить тип *real* на тип *double*, то наиболее просто это можно сделать, поместив в начале программы следующее описание:

```
type
real=double;
```

## Контрольные вопросы по главе 1

1) Что такое алгоритм? Приведите известные вам определения алгоритма.
2) Приведите основные признаки алгоритма.
3) Что такое определенность, выполнимость и конечность алгоритма?
4) Какие вы знаете основные классы алгоритмов?
5) Что такое структурное программирование?
6) Приведите основной список инструкций, используемых при структурном программировании.
7) Какие принципы вы можете порекомендовать при написании программ?
8) Что такое документированность программы?
9) Что такое рекуррентные соотношения и рекуррентные алгоритмы?
10) Что такое схема программы?
11) Что такое рекурсия и когда ее не следует применять?
12) Как алгоритмы связаны с используемыми структурами данных?

# Глава 2

# ИНФОРМАТИКА И ЭЛЕКТРИЧЕСКИЕ ЦЕПИ

## 2.1 Модель цепи в пространстве состояний

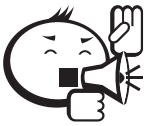
Электрическая цепь, содержащая $n$ реактивных элементов — индуктивностей и/или емкостей, — может быть представлена в виде системы $n$ линейных дифференциальных уравнений 1-го порядка с постоянными коэффициентами. Этими дифференциальными уравнениями связываются токи через индуктивности, напряжения на емкостях и возбуждающие сигналы, в роли которых выступают изменяющиеся во времени величины ЭДС источников напряжения и/или величины токов источников тока.

В цепях, содержащих реактивные элементы разного типа (емкости и индуктивности), возможны колебательные переходные процессы, связанные с циклическим перетоком энергии между емкостью и индуктивностью. В этом случае можно говорить о частотах собственных колебаний системы и о *резонансных явлениях*, характеризующихся возрастанием амплитуды колебаний токов/напряжений в цепи при ее возбуждении периодическим внешним воздействием, частота которого близка к частоте ее собственных колебаний.

Модель цепи в виде системы дифференциальных уравнений относительно токов, протекающих через индуктивности, и напряжений на емкостях называется *моделью электрической цепи в пространстве состояний*. Выбор такого состава компонент пространства состояний обусловлен тем, что токи индуктивностей и напряжения емкостей не могут изменяться скачком при любых коммутациях. Вследствие этого траектории движения системы во времени в таком пространстве оказываются непрерывными, и конечная точка одного фрагмента траектории служит начальной точкой для последующего.

## 2.2 Получение модели цепи в пространстве состояний на основе системы уравнений Кирхгофа

Любая математическая модель строится на основе формальной записи физических закономерностей. Для электрической цепи физические закономерности отражаются *законами Кирхгофа* с использованием т. н. *компонентных соотношений*.



Напомним, что первый закон Кирхгофа «запрещает» накопление зарядов в узлах схемы: *сумма токов, втекающих в узел, равна сумме токов, вытекающих из узла*.

Второй закон Кирхгофа представляет собой одну из форм закона сохранения энергии: работа, совершаемая электрическими силами при перемещении заряда по любому замкнутому контуру в цепи, равна работе внешних сил, разделяющих заряды в источниках. Или, что эквивалентно, *алгебраическая сумма падений напряжений на участках контура равна сумме алгебраической суммы ЭДС источников, входящих в этот контур*.

Компонентные соотношения связывают напряжения на элементе цепи с величиной протекающего через него тока.

Для активных сопротивлений компонентные соотношения отражают известное выражение закона Ома для участка цепи:

$$i = \frac{U}{R}, \tag{2.1}$$

где $i$ — ток через сопротивление; $U$ — падение напряжения на сопротивлении; $R$ — величина сопротивления.

Для реактивных элементов (индуктивностей и емкостей) законы связи токов и напряжений носят дифференциальный характер.

Для индуктивности:

$$U_L = L\frac{di_L}{dt}, \tag{2.2}$$

где $i_L$ — ток через индуктивность; $U_L$ — падение напряжения на индуктивности; $L$ — величина индуктивности.

Для емкости:

$$i_C = C\frac{dU_C}{dt}, \tag{2.3}$$

где $i_C$ — ток через емкость; $U_C$ — падение напряжения на емкости; $C$ — величина емкости.

Система уравнений Кирхгофа позволяет связать напряжения и токи в цепи алгебраическими уравнениями. Для линейных цепей, параметры которых $R$, $L$ и $C$ не зависят ни от величин токов, протекающих через элементы, ни от напряжений на них, эти уравнения имеют линейный вид. Наличие дифференциальных связей (2.2) и (2.3) приводит к тому, что часть алгебраических уравнений превращаются в дифференциальные.

## 2.3 Пример построения модели цепи в пространстве состояний

Пусть имеется схема цепи с одним внешним воздействием в виде источника ЭДС $E(t)$, показанная на рисунке 2.1.

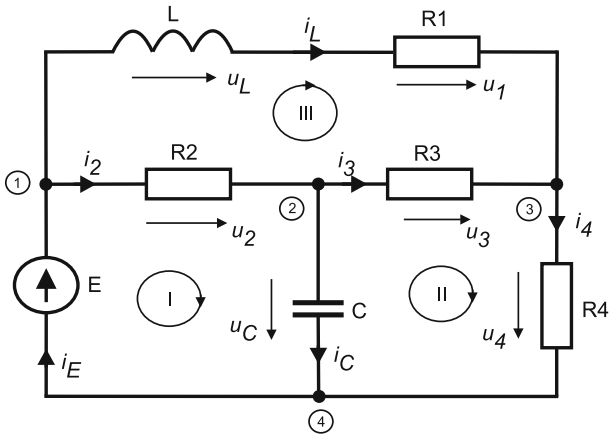



Рис. 2.1 – Схема цепи

В схеме имеется 6 ветвей и 4 узла, после разметки выделено 3 контура.

Из курса ТОЭ известно, что для такой цепи можно составить 6 уравнений: 3 для баланса токов в узлах (I закон Кирхгофа) и 3 для баланса напряжений в контурах (II закон Кирхгофа).

Запишем систему уравнений Кирхгофа для этой цепи с учетом компонентных соотношений.

$$
\begin{aligned}
&1)\ \text{для узла 1: } i_E - i_2 - i_L = 0; \qquad (2.4)\\
&2)\ \text{для узла 2: } i_2 - C\frac{dU_C}{dt} - i_3 = 0;\\
&3)\ \text{для узла 3: } i_L + i_3 - i_4 = 0;\\
&4)\ \text{для контура I: } R2 \cdot i_2 + U_C - E = 0;\\
&5)\ \text{для контура II: } -U_C + R3 \cdot i_3 + R4 \cdot i_4 = 0;\\
&6)\ \text{для контура III: } L\frac{di_L}{dt} + R1 \cdot i_L - R4 \cdot i_4 - R3 \cdot i_3 = 0.
\end{aligned}
$$

Система уравнений (2.4) связывает 9 переменных:

$$\frac{di_L}{dt},\ \frac{dU_C}{dt},\ i_2,\ i_3,\ i_4,\ i_E,\ i_L,\ U_C,\ E.$$

При правильной записи уравнений Кирхгофа получается система независимых уравнений, и можно выразить любые 6 переменных через оставшиеся 3.

Поставим задачу выразить переменные $\frac{di_L}{dt},\ \frac{dU_C}{dt},\ i_2,\ i_3,\ i_4,\ i_E$ через $i_L,\ U_C,\ E$.

Воспользуемся для такого выражения средствами матричной алгебры. Представим систему (2.4) в матричной форме:

$$X \cdot D = 0, \qquad (2.5)$$

где $D$ — прямоугольная матрица $9 \times 6$ (9 столбцов и 6 строк); $X$ — вектор-столбец из 9 компонент.

Для нашего примера матрица $D$ и вектор $X$ будут иметь вид:

$$D = \begin{pmatrix} 0 & 0 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -C & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & R2 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & R3 & R4 & 0 & -1 & 0 \\ L & 0 & 0 & -R2 & -R3 & 0 & R1 & 0 & 0 \end{pmatrix} \quad X = \begin{pmatrix} \frac{di_L}{dt} \\ \frac{dU_C}{dt} \\ i_E \\ i_2 \\ i_3 \\ i_4 \\ i_L \\ U_C \\ E \end{pmatrix}. \tag{2.6}$$

Преобразуем (2.5) к такому виду:

$$D0 \cdot \begin{pmatrix} \frac{di_L}{dt} \\ \frac{dU_C}{dt} \\ i_E \\ i_2 \\ i_3 \\ i_4 \end{pmatrix} = D1 \cdot \begin{pmatrix} i_L \\ U_C \\ E \end{pmatrix}, \tag{2.7}$$

где матрицы $D0$ и $D1$ для рассматриваемого примера имеют вид:

$$D0 = \begin{pmatrix} 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & -C & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & R2 & 0 & 0 \\ 0 & 0 & 0 & 0 & R3 & R4 \\ L & 0 & 0 & -R2 & -R3 & 0 \end{pmatrix} \quad D1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 1 & 0 \\ -R1 & 0 & 0 \end{pmatrix}. \tag{2.8}$$

Ввиду того, что исходная система уравнений линейно независима, матрица $D0$ имеет обратную, поставленная выше задача имеет единственное решение:

$$\begin{pmatrix} \frac{di_L}{dt} \\ \frac{dU_C}{dt} \\ i_E \\ i_2 \\ i_3 \\ i_4 \end{pmatrix} = G \cdot \begin{pmatrix} i_L \\ U_C \\ E \end{pmatrix}, \tag{2.9}$$

где $G = D0^{-1} \cdot D1$ — матрица $6 \times 3$ (6 строк и 3 столбца).

В нашем случае матрица $G$ имеет вид:

$$G = \begin{pmatrix} \boxed{\begin{matrix} \dfrac{-(R3\cdot R4 + R1\cdot R4 + R1\cdot R3)}{(R4+R3)\cdot L} & \dfrac{-R4}{(R4+R3)\cdot L} \\ \dfrac{R4}{C\cdot(R4+R3)} & \dfrac{-(R4+R3+R2)}{C\cdot R2\cdot(R4+R3)} \end{matrix}} & \boxed{\begin{matrix} \dfrac{1}{L} \\ \dfrac{1}{C\cdot R2} \end{matrix}} \\ \begin{matrix} 1 \\ 0 \\ \dfrac{-R4}{R4+R3} \\ \dfrac{R3}{R4+R3} \end{matrix} & \begin{matrix} \dfrac{-1}{R2} \\ \dfrac{-1}{R2} \\ \dfrac{1}{R4+R3} \\ \dfrac{1}{R4+R3} \end{matrix} \quad \begin{matrix} \dfrac{1}{R2} \\ \dfrac{1}{R2} \\ 0 \\ 0 \end{matrix} \end{pmatrix}. \qquad (2.10)$$

В системе уравнений (2.9) первые два уравнения дифференциальные, остальные — алгебраические. В выражении (2.10) прямоугольниками выделены компоненты матрицы $G$, представляющие параметры системы дифференциальных уравнений, — модель цепи в пространстве состояний. Выпишем в явной форме систему дифференциальных уравнений:

$$\begin{pmatrix} \dfrac{di_L}{dt} \\ \dfrac{dU_C}{dt} \end{pmatrix} = A\cdot\begin{pmatrix} i_L \\ U_C \end{pmatrix} + b\cdot E(t), \qquad (2.11)$$

где

$$A = \begin{pmatrix} \dfrac{-(R3\cdot R4 + R1\cdot R4 + R1\cdot R3)}{L\cdot(R4+R3)} & \dfrac{-R4}{L\cdot(R4+R3)} \\ \dfrac{R4}{C\cdot(R4+R3)} & \dfrac{-(R4+R3+R2)}{C\cdot(R4+R3)\cdot R2} \end{pmatrix} \quad b = \begin{pmatrix} \dfrac{1}{L} \\ \dfrac{1}{C\cdot R2} \end{pmatrix}.$$

Решение уравнений (2.11) позволяет найти процессы $i_L(t)$ и $U_C(t)$. Оставшиеся алгебраические уравнения системы (2.9) дают выражения токов в ветвях схемы в виде линейной комбинации процессов $i_L(t)$, $U_C(t)$ и $E(t)$.

Как правило, модель цепи в пространстве состояния помимо дифференциальных уравнений (ДУ), определяющих ее динамику, включает еще и т. н. *уравнения наблюдения*.

***Уравнения наблюдения*** *— это выражения токов, протекающих через какие-либо заданные элементы, и/или напряжений между заданными точками схемы.*

Пусть в нашем примере наблюдаемыми величинами являются ток через емкость и напряжение на резисторе $R4$. Согласно компонентным соотношениям

$$U_4 = R4\cdot i_4 \text{ и } i_C = C\cdot\frac{dU_C}{dt}.$$

Получим выражения для наблюдаемых переменных $U_{R4}$ и $i_C$, используя выражения для $i_4$ и $\frac{dU_C}{dt}$ из (2.9):

$$U_4(t) = R4 \cdot \left( \frac{R3}{R4 + R3} \cdot i_L(t) + \frac{1}{R4 + R3} \cdot U_C(t) + 0 \cdot E(t) \right); \tag{2.12}$$

$$i_C(t) = C \cdot \left( \frac{R4}{C \cdot (R4 + R3)} \cdot i_L(t) - \frac{R4 + R3 + R2}{C \cdot R2 \cdot (R4 + R3)} \cdot U_C(t) + \frac{1}{C \cdot R2} \cdot E(t) \right)$$

или в матричной форме:

$$\begin{pmatrix} U_4(t) \\ i_C(t) \end{pmatrix} = \begin{pmatrix} \frac{R4 \cdot R3}{R4 + R3} & \frac{R4}{R4 + R3} & 0 \\ \frac{R4}{R4 + R3} & -\frac{R4 + R3 + R2}{R4 \cdot (R4 + R3)} & \frac{1}{R2} \end{pmatrix} \cdot \begin{pmatrix} i_L(t) \\ U_C(t) \\ E(t) \end{pmatrix}. \tag{2.13}$$

Уравнения (2.11–2.13) представляют собой полную модель цепи в пространстве состояний. Дальнейшее решение задачи сводится к отысканию решения системы ДУ (2.11).

## 2.4 Проблема вычислений

Из приведенного примера видно, что, помимо знаний правил построения электрических цепей, для окончательного решения задачи требуется знание математического аппарата в области матричных опреаций, систем линейных уравнений, диффиренциальных уравнений, систем ДУ и т. д. Усложнение состава электрической цепи влечет за собой существенное усложнение математической модели. Аналитические вычисления становятся очень трудоемкими, даже при условии отсутствия ошибок как в исходных данных так и в процессе вычисления. Зачастую найти аналитическое решение, особенно в части решения ДУ и систем ДУ, не представляется возможным.

Имея под рукой огромные вычислительные мощности современных персональных компьютеров, процесс вычисления можно сушественно ускорить, переложив на него рутинную работу. На сегодняшний момент разработано множество методов решения математических задач различного уровня с использованием ПК. Процесс моделирования упрощается. Достаточно изменить какой-либо входной параметр и запустить программу вычислений. Результат будет получен за несколько минут, а не часов, как в случае если бы расчеты проводились вручную.

Конечно, возможно (и разумно) строить модель различными средствами облегчающими математические вычисления. Но, как уже отмечалось ранее, порой различные прикладные программы недоступны, в силу различных лицензионных соглашений, стоимости и т. д.

Возикающие задачи, хоть и кажутся на первый взгляд очень сложными, при более пристальном рассмотрении оказываются существенно проще. Для их решения достаточно навыка программирования и знания алгоритмов численного решения математических задач.

## Контрольные вопросы по главе 2

1) Что собой представляет модель электрической цепи в пространстве состояний?
2) Почему при построении модели электрической цепи появляются дифференциальные уравнения?
3) Какой качественный смысл несут зачения элементов матрицы коэффициентов матричного уравнения, описывающего модель электрической цепи.

# Глава 3

# ЧИСЛЕННЫЕ АЛГОРИТМЫ

Начиная с глубокой древности, людям приходится заниматься вычислениями. Одной из первых вычислительных формул является хорошо известная теорема Пифагора. По крайней мере, сотни лет в навигации и геодезии используются различные математические таблицы. Однако только после появления компьютеров численный анализ окончательно сформировался как самостоятельная наука, а крупномасштабные автоматизированные вычисления стали играть большую роль в науке и технике. Появление компьютеров изменило характер вычислений по двум причинам — во-первых, резко вырос объем вычислений, во-вторых, качественно изменился их характер, т. к. машинная арифметика существенно отличается от арифметики ручной.

*Это отличие объясняется, в основном, следующими причинами:*

1) множество *чисел с плавающей точкой* (т. е. вещественных чисел, имеющих *различное* машинное представление) конечно;
2) существуют наименьшие и наибольшие *числа с плавающей точкой*;
3) существует вещественное число (машинная точность), равное разнице между 1.0 и следующим по величине вещественным числом;
4) арифметические операции редко приводят к точно представимым результатам, поэтому результат округляется до ближайшего *числа с плавающей точкой*;
5) *числа с плавающей точкой* расположены между нулем и наибольшим значением неравномерно — их больше около нуля.

Компьютерные вычисления почти всегда происходят с ошибками, причем особую роль играют даже не ошибки программиста, а ошибки во входных данных (связанные, например, с погрешностями измерений) и, особенно, ошибки округления и ошибки при формулировании корректного численного алгоритма. Последние два типа ошибок связаны, в основном, с устойчивостью и чувствительностью алгоритма к малым ошибкам.

# 3.1 Решение систем линейных уравнений

Одна из задач, наиболее часто встречающихся в научных и прикладных вычислениях, — решение *системы линейных уравнений*; при этом обычно число уравнений равно числу неизвестных. Такую систему можно записать в виде

$$A \times \mathbf{x} = \mathbf{b},$$

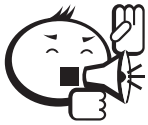
где $A$ — заданная квадратная матрица порядка $n$, $\mathbf{b}$ — заданный вектор-столбец с $n$ компонентами и $\mathbf{x}$ — неизвестный вектор-столбец с $n$ компонентами. Иными словами, $A = \{a_{ij}\}$, $i, j = 1, \ldots, n$, $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. То же самое можно записать в *координатной* форме:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1; \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2; \\ \qquad\qquad \ldots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n = b_n; \end{cases} \tag{3.1}$$

или $\sum_{j=1}^{n} a_{ij}x_j = b_i$, $i = 1, \ldots, n$.

Среди источников линейных уравнений необходимо отметить аппроксимацию дифференциальных уравнений конечными (дискретными) алгебраическими системами, построение полиномов или кривых какого-либо иного специального вида по заданной информации, а также множество других задач.

При изучении линейной алгебры студенты, как правило, в первую очередь знакомятся с решением невырожденных систем линейных уравнений при помощи правила Крамера, согласно которому все компоненты решения представляются отношениями определителей с различными числителями и общим знаменателем. Однако если бы вы попробовали решить систему из 20 уравнений с помощью правила Крамера, то вам потребовалось бы вычислить 21 определитель порядка 20, $20 \times 21$ определитель порядка 19 и т. д. В итоге, решение линейной системы предполагает $21 \times 20! \times 19$ умножений плюс примерно такое же число сложений. На любом современном быстродействующем компьютере для этого потребуется многие тысячи лет, при условии, что машина не выйдет из строя в процессе вычислений.



Для больших систем уравнений правило Крамера обычно ведет к чрезмерным ошибкам округлений.

Из линейной алгебры известно также, что решение системы $A \times x = b$ можно представить в виде $\mathbf{x} = A^{-1} \times \mathbf{b}$, где $A^{-1}$ — матрица, обратная для $A$. Однако в огромном большинстве практических вычислительных задач вовсе не обязательно и даже нежелательно в действительности находить $A^{-1}$. Этот метод дает не слишком точный результат и, главное, требует выполнения слишком большого количества лишних операций. Вследствие этого обратим главное внимание на **прямое** решение систем уравнений, а не на вычисление обратной матрицы.

Важно различать два типа матриц:

1) Хранимая матрица, т. е. матрица, все $n^2$ элементов которой хранятся в оперативной памяти машины.

2) Разреженная матрица, т. е. матрица, большинство элементов которой нули, а ненулевые элементы могут или храниться посредством какой-либо специальной структуры данных, или генерироваться по мере необходимости. Матрицы этого типа часто встречаются при решении дифференциальных уравнений в частных производных конечно-разностными и конечно-элементными методами. При этом порядок $n$ может достигать нескольких десятков тысяч, а иногда и того больше. Среди разреженных матриц особое значение имеют *ленточные матрицы* — матрицы, все ненулевые элементы которых расположены вблизи главной диагонали, т. е. $a_{ij} = 0$ для всех $i$, $j$, таких, что $|i - j| > m$, причем, как правило, $m << n$. Величина $2m + 1$ называется *шириной ленты*, все ненулевые элементы расположены на главной диагонали, на $m$ диагоналях, лежащих выше главной диагонали, и на $m$ диагоналях, лежащих ниже ее. Ленточные матрицы важны не только потому, что они часто появляются в приложениях, но и потому что для них существуют специальные эффективные алгоритмы. Ниже приведен пример *трехдиагональной* ленточной матрицы. Понятно, что для хранения этой матрицы в памяти компьютера необходимо только один массив размерностью $n$ и два массива размерностью $n - 1$ (или три массива размерностью $n$).

Пример

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \end{pmatrix}.$$

Разреженные и хранимые матрицы в известной степени пересекаются. Хранимая матрица может иметь много нулевых элементов и, следовательно, быть в то же время разреженной; однако если для нулевых элементов отводится место в оперативной памяти и разреженность матрицы не учитывается, то, как правило, ее разреженной не называют.

Некоторые численные методы, применяемые для хранимых матриц, весьма отличаются от методов, приспособленных для разреженных матриц.

Методы для хранимых матриц более универсальны, и им будет уделено основное внимание. Эти методы можно модифицировать таким образом, чтобы обрабатывать ленточные матрицы и другие типы больших или умеренно разреженных матриц.

### 3.1.1 Метод Гаусса

Рассмотрим решение линейной системы алгебраических уравнений:

$$A \times \mathbf{x} = \mathbf{b},$$

с хранимой $n \times n$-матрицей $A$ и векторами **b** и **x** порядка $n$. Для решения этой задачи обычно применяется алгоритм, являющийся одним из старейших численных методов, — метод последовательного исключения неизвестных, называемый методом Гаусса. Он основан на приведении матрицы системы к треугольному виду путем последовательного исключения неизвестных из уравнений системы.

Предположим, что $a_{11} \neq 0$. Разделив первое уравнение (3.1) на $a_{11}$, получим:

$$x_1 = b_1^{(1)} - \sum_{j=2}^{n} a_{1j}^{(1)} x_j; \quad a_{1j}^{(1)} = \frac{a_{1j}}{a_{11}}; \quad b_1^{(1)} = \frac{b_1}{a_{11}}.$$

Затем полученое $x_1$ подставляется в оставшиеся $n - 1$ уравнений. В результате получается система $n - 1$ уравнений с $n - 1$ неизвестными $x_2, \ldots, x_n$.

$$\sum_{j=2}^{n} a_{ij}^{(1)} x_j = b_i^{(1)}; \quad a_{ij}^{(1)} = a_{ij} - a_{i1}\frac{a_{1j}}{a_{11}}; \quad b_i^{(1)} = b_i - a_{i1}\frac{b_i}{a_{11}}; \quad i, j = 2, 3, \ldots, n.$$

Этот процесс называется исключением неизвестного. Если его повторить $n - 1$ раз, то система сведется к одному уравнению с одним неизвестным, а такое уравнение решается совсем просто. Чтобы получить программу, точно отражающую этот процесс, мы в общем виде сформулируем $k$-й шаг исключения неизвестного. На $k$-м шаге мы имеем дело с $n - k + 1$ линейными уравнениями.

$$\sum_{j=k}^{n} a_{ij}^{(k)} x_j = b_i^{(k)}, \; i = k \ldots n.$$

Новые коэффициенты $a_{ij}^{(k+1)}$ и $b_{ij}^{(k+1)}$ вычисляются так, чтобы получилась система $n - k$ уравнений:

$$\sum_{j=k+1}^{n} a_{ij}^{(k+1)} x_j = b_i^{(k+1)}, \; i = k + 1 \ldots n.$$

Эти коэффициенты получаются как линейная комбинация коэффициентов $k$-го и $i$-го уравнений (строк)

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \left(\frac{a_{kj}^{(k)}}{a_{kk}^{(k)}}\right) a_{ik}^{(k)}, \tag{3.2}$$

$$b_i^{(k+1)} = b_i^{(k)} - \left(\frac{b_k^{(k)}}{a_{kk}^{(k)}}\right) a_{ik}^{(k)},$$

для $i, j = k + 1, \ldots, n$. Теперь $k$-е уравнение вычитается из $i$-го, предварительно умноженое на величину, выбранную так, чтобы для $j = k$ и соответствующего $i$

$$a_{ik}^{(k+1)} = a_{ik}^{(k)} - \left( \frac{a_{kk}^{(k)}}{a_{kk}^{(k)}} \right) a_{ik}^{(k)} = 0.$$

Это значит, что в новой системе все коэффициенты при $x_k$ равны нулю и, таким образом, исключается $x_k$. Заметим, кстати, что нет необходимости вычислять коэффициенты $a_{ik}^{(k+1)}$.

После $n - 1$ шагов система сведется к уравнению

$$a_{nn}^{(n)} x_n = b_n^{(n)},$$

из которого непосредственно вычисляется $x_n$. Остальные неизвестные получаются подстановкой уже вычисленных неизвестных в полученные ранее уравнения. Например, для вычисления $x_{n-1}$ достаточно подставить $x_n$ в

$$a_{n-1,n-1}^{(n-1)} x_{n-1} + a_{n-1,n}^{(n-1)} x_n = b_{n-1}^{(n-1)}.$$

Этот процесс называется *обратной подстановкой*. Обратную подстановку на $k$-м шаге можно выразить следующей общей формулой

$$x_k = \frac{b_i^k - \sum_{j=k+1}^{n} a_{ij}^{(k)} x_j}{a_{ik}^{(k)}} \tag{3.3}$$

для произвольного $i$, такого, что $k \leqslant i \leqslant n$.

Обратите внимание на последовательность, в которой выполняются обратные подстановки. Она определяется тем, что для вычисления $x_k$ должны быть известны значения $x_{k+1}, \ldots, x_n$.

## 3.1.2 Обусловленность матрицы

Ошибки округлений, совершенных в процессе вычислений, почти всегда приводят к тому, что вычисленное решение (которое мы будем теперь обозначать через $\mathbf{x}_*$) в определенной степени отличается от теоретического решения $\mathbf{x} = A^{-1}\mathbf{b}$. В действительности, оно и должно отличаться, поскольку компоненты вектора $x$ обычно не являются числами с плавающей точкой.

*Имеются две общепринятые меры отклонения для* $\boldsymbol{x}_*$*:*

***ошибка***

$$e = \boldsymbol{x} - \boldsymbol{x}_*$$

*и*

***невязка***

$$r = \boldsymbol{b} - A\boldsymbol{x}_*.$$

Из теории матриц известно, что если одна из этих величин равна нулю, то и другая также должна быть равна нулю. Но они совсем не обязательно одновременно будут «малыми».

## Пример

Рассмотрим следующий простой (хотя и специально сконструированный) пример:

$$\begin{pmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.217 \\ 0.254 \end{pmatrix}.$$

Если провести Гауссово исключение с частичным выбором на гипотетической *трехзначной* десятичной машине (т. е. результатом любой арифметической операции является число с тремя десятичными значащими цифрами), то получим решение

$$\mathbf{x}_* = \begin{pmatrix} -0.443 \\ 1.000 \end{pmatrix}.$$

Если бы мы не знали правильного ответа, то с целью оценить полученную точность могли бы (точно) вычислить невязку:

$$r = \mathbf{b} - A\mathbf{x}_* = \begin{pmatrix} -0.000460 \\ -0.000541 \end{pmatrix}.$$

Все невязки меньше, чем $10^{-3}$. Едва ли можно ожидать лучшего на трехзначной машине. Однако легко видеть, что точное решение этой системы есть

$$\mathbf{x} = \begin{pmatrix} 1.000 \\ -1.000 \end{pmatrix}.$$

Таким образом, ошибка больше, чем решение.

Конечно приведенный пример специально сконструирован и нетипичен, однако Гауссово исключение с частичным выбором ведущего элемента гарантированно дает малые невязки. Связь между величиной невязки и величиной ошибки определяется отчасти характеристикой, называемой числом обусловленности матрицы.

Коэффициенты матрицы и правой части системы линейных уравнений редко бывают известны точно. Некоторые системы возникают из эксперимента, и тогда коэффициенты подвержены ошибкам наблюдения. Коэффициенты других систем записываются формулами, что влечет ошибки округлений при их вычислении. Даже если систему можно точно записать в память машины, в ходе ее решения почти неизбежно будут сделаны ошибки округлений. Можно показать, что ошибки округлений в Гауссовом исключении имеют то же влияние на ответ, что и ошибки в исходных коэффициентах.

Вследствие этого мы подходим к фундаментальному вопросу. Если в коэффициентах системы линейных уравнений делаются ошибки, то как сильно при этом меняется решение? Или, другими словами, если $A \times \mathbf{x} = \mathbf{b}$, то как можно измерить чувствительность $\mathbf{x}$ по отношению к изменениям в $A$ и $\mathbf{b}$? Ответ на этот вопрос лежит в уточнении понятия «почти вырожденная». Если $A$ — вырожденная матрица, то для некоторых $\mathbf{b}$ решение $\mathbf{x}$ не существует, тогда как для других $\mathbf{b}$ оно будет неединственным. Таким образом, если $A$ почти вырождена, то можно ожидать, что малые изменения в $A$ и $\mathbf{b}$ вызовут очень большие изменения в $\mathbf{x}$. С другой стороны, если $A$ — единичная матрица, то $\mathbf{b}$ и $\mathbf{x}$ — один и тот же вектор. Следовательно, если $A$ близка к единичной матрице, то малые изменения в $A$ и $\mathbf{b}$ должны влечь за собой соответственно малые изменения в $\mathbf{x}$.

На первый взгляд может показаться, что есть некоторая связь между величиной ведущих элементов в Гауссовом исключении с частичным выбором и близостью к вырожденности, поскольку если бы арифметику можно было выполнять точно, то все ведущие элементы были бы отличны от нуля тогда и только тогда, когда матрица не вырождена. До некоторой степени верно также, что если ведущие элементы малы, то матрица близка к вырожденной. Однако при наличии ошибок округления обратное уже неверно, матрица $\mathbf{x}$ может быть близка к вырожденной даже если ни один из ведущих элементов не мал.

Чтобы получить более точную и надежную меру близости к вырожденности, нам потребуется ввести понятие нормы вектора [6]. Норма — это число, которое измеряет общий уровень элементов вектора. Наиболее употребительной векторной нормой является евклидова длина

$$\left(\sum_{i=1}^{n}|x_i|^2\right)^{\frac{1}{2}}.$$

Согласованной с ней нормой в пространстве матриц является норма:

$$\|A\| = \sqrt{\max_i \lambda^i_{A^T A}}.$$

Однако использование этой нормы делает слишком трудоемкими многие вычисления. Вместо нее определим норму вектора из $n$ компонент следующим образом:

$$\|\mathbf{x}\| = \sum_{i=1}^{n}|x_i|.$$

Согласованной с ней нормой в пространстве матриц является норма:

$$\|A\| = \max_j \left( \sum_i |a_{ij}| \right).$$

Эта норма обладает многими из аналитических свойств евклидовой длины. Некоторые из геометрических свойств евклидовой длины теряются, но они не слишком важны здесь.

Умножение вектора $\mathbf{x}$ на матрицу $A$ приводит к новому вектору $A \times \mathbf{x}$, норма которого может очень отличаться от нормы вектора $\mathbf{x}$. Это изменение нормы прямо связано с той чувствительностью, которую мы хотим измерять. Область возможных изменений может быть задана двумя числами:

$$M = \max_{\mathbf{x}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|},$$
$$m = \min_{\mathbf{x}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Максимум и минимум берутся по всем ненулевым векторам. Заметим, что если $A$ вырождена, то $m = 0$. Отношение $M/m$ называется числом обусловленности матрицы $A$,

$$\mathrm{cond}(A) = \frac{\max\limits_{\mathbf{x}} \dfrac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}}{\min\limits_{\mathbf{x}} \dfrac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}}.$$

Пусть $\Delta\mathbf{b}$ — ошибка в определении $\mathbf{b}$, а $\Delta\mathbf{x}$ — соответствующая ей ошибка в решении $\mathbf{x}$. Тогда можно показать, что

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leqslant \mathrm{cond}(A) \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

Это неравенство показывает, что число обусловленности выполняет роль множителя в увеличении относительной ошибки. Изменения правой части могут повлечь за собой изменения в решении, большие в $\mathrm{cond}(A)$ раз. То же самое справедливо в отношении изменений в коэффициентах матрицы.

Число обусловленности является также мерой близости к вырожденности. Хотя мы не имеем еще математических средств, необходимых для точной формулировки этого утверждения, можно рассматривать число обусловленности как величину, обратную к относительному расстоянию от данной матрицы до множества вырожденных матриц. Число обусловленности также играет фундаментальную роль в анализе ошибок округлений, совершенных в процессе Гауссова исключения.

Реальное вычисление числа $\mathrm{cond}(A)$ предполагает знание $A^{-1}$. Если $a_j$ — столбцы $A$, а $a_j^*$ столбцы $A^{-1}$, то для векторной нормы, которую мы используем,

$$\mathrm{cond}(A) = \max_j \|\mathbf{a}_j\| \max_j \|\mathbf{a}_j^*\|.$$

Легко вычислить $\|A\|$, однако вычисление $\|A^{-1}\|$ примерно удваивает время, нужное для Гауссова исключения. К счастью, при решении систем линейных алгебраических уравнений точное значение числа обусловленности не требуется, т. к. обычно бывает достаточно любой разумной приближенной оценки для него.

Для плохо обусловленных матриц метод Гауссова исключения дает, как мы видели, большие ошибки. Поэтому хорошие процедуры решения систем линейных уравнений должны оценивать число обусловленности матрицы некоторым образом. Например, для этого можно использовать следующее приближенное соотношение:

$$\mathrm{cond}(A) \approx \max_j \|\mathbf{a}_j\| \frac{\|\mathbf{z}\|}{\|\mathbf{y}\|},$$

где $\mathbf{y}$ и $\mathbf{z}$ — векторы, определяемые таким образом, что $\|\mathbf{z}\|/\|\mathbf{y}\| \approx \|A^{-1}\|$. Для этого решаются две системы уравнений:

$$A^T\mathbf{y} = \mathbf{e},$$
$$A\mathbf{z} = \mathbf{y},$$

где $A^T$ — транспонированная для матрицы $A$, а $\mathbf{e}$ — вектор с компонентами $\pm 1$, выбираемый так, чтобы по возможности увеличить (максимизировать) $\mathbf{y}$.

Эта оценка является лишь нижней границей для действительного числа обусловленности, однако есть определенные теоретические основания ожидать, что это очень точная оценка.

### 3.1.3 Большие разреженные системы

Многие линейные алгебраические системы имеют столь большое число $n$ уравнений и неизвестных, что хранить в оперативной памяти компьютера полную квадратную матрицу из $n$ элементов просто невозможно. Обычно такие системы получаются при дискретизации дифференциальных уравнений, обыкновенных или с частными производными, а также при расчете различных конструкций или цепей [8]. Зачастую матрицы таких задач настолько разрежены, что памяти компьютера хватает только для хранения всех ненулевых элементов вместе с информацией об их расположении. Как же нужно решать соответствующую систему линейных уравнений?

Если применение Гауссова исключения возможно, то оно остается очень экономичным, точным и полезным алгоритмом. Исключение возможно, если удается разместить в памяти ненулевые элементы треугольных матриц, строящихся в ходе исключения, вместе с информацией о том, где они помещены. В противном случае предпочтительными являются итерационные методы.

#### Метод прогонки

Рассмотренный выше метод решения линейных систем уравнений, ориентированный на матрицы самого общего вида, окажется крайне неэффективным при его

использовании для решения систем с трехдиагональными матрицами. Большую часть времени компьютер будет заниматься обработкой нулевых элементов. Поэтому для решения таких систем используется модификация метода исключения Гаусса, в которой учитывается структура матрицы $A$. Этот алгоритм называется *методом прогонки*.

Пусть дана система линейных алгебраических уравнений вида:

$$\begin{aligned} &y_0 + \lambda_1 y_1 = \mu_1; \\ &a_i y_{i-1} + c_i y_i + b_i y_{i+1} = f_i; \quad i = 1 \ldots n-1, \\ &y_n + \lambda_2 y_{n-1} = \mu_2. \end{aligned} \tag{3.4}$$

Это обычный вид таких систем, которые появляются при решении практических задач. Здесь $\lambda$, $\mu$, $a$, $b$, $c$ — константы.

Запишем ее в развернутом виде.

$$\begin{pmatrix} 1 & \lambda_1 & 0 & 0 & \ldots & 0 & 0 \\ a_1 & c_1 & b_1 & 0 & \ldots & 0 & 0 \\ 0 & a_2 & c_2 & b_2 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & a_{n-1} & c_{n-1} & b_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & \lambda_2 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \ldots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} \mu_1 \\ f_1 \\ f_2 \\ \ldots \\ f_{n-1} \\ \mu_2 \end{pmatrix}.$$

Будем искать решение этой системы в виде рекуррентной формулы, связывающей $y_i$ и $y_{i+1}$:

$$y_i = \alpha_{i+1} y_{i+1} + \beta_{i+1}, \; i = 0, 1, \ldots, n-1, \tag{3.5}$$

где $\alpha_1$ и $\beta_1$ — известные константы ($\alpha_1 \equiv -\lambda_1$; $\beta_1 \equiv \mu_1$), а $\alpha_2, \ldots, \alpha_n$ и $\beta_2, \ldots, \beta_n$ — пока неизвестные константы.

Понизим в (3.5) индекс на 1:

$$y_{i-1} = \alpha_i y_i + \beta_i. \tag{3.6}$$

Теперь, подставив (3.6) во второе уравнение системы (3.4):

$$\alpha_i(\alpha_i y_i + \beta_i) + c_i y_i + b_i y_{i+1} = f_i,$$

где $i = 1, \ldots, n-1$, приходим к уравнению вида

$$y_i + \frac{b_i}{a_i\alpha_i + c_i} y_{i+1} = \frac{f_i - \alpha_i\beta_i}{a_i\alpha_i + c_i}.$$

Сравнивая полученное равенство с (3.5), находим

$$\alpha_{i+1} = \frac{-b_i}{a_i\alpha_i + c_i}; \quad \beta_{i+1} = \frac{f_i - \alpha_i\beta_i}{a_i\alpha_i + c_i}. \tag{3.7}$$

Эти рекуррентные соотношения называются *формулами прямой прогонки*. Они позволяют по заданным значениям $\alpha_1$ и $\beta_1$ последовательно определить все пары прогоночных коэффициентов $(\alpha_2, \beta_2)$, $(\alpha_3, \beta_3)$, …, $(\alpha_n, \beta_n)$.

Затем начинается второй этап метода прогонки — обратная прогонка.

Запишем уравнение (3.5) для $i = n - 1$ и добавим еще не использованное третье уравнение системы (3.4). Получаем систему из двух уравнений с двумя неизвестными $y_n$ и $y_{n-1}$:

$$\begin{cases} y_{n-1} = \alpha_n y_n + \beta_n; \\ y_n = -\lambda_2 y_{n-1} + \mu_2. \end{cases}$$

Ее решение:

$$y_n = \frac{\lambda_2 \beta_n + \mu_2}{1 + \lambda_2 \alpha_n}. \tag{3.8}$$

Все остальные значения $y_i$ ($i = n - 1, n - 2, \ldots, 0$) находим по рекуррентной формуле (3.5). Формулы (3.5) и (3.8) называются *формулами обратной прогонки*. Метод прогонки является одним из наиболее эффективных и распространенных численных алгоритмов линейной алгебры.

## Итерационные методы

Имеется важный класс систем линейных уравнений, для которых элементы матрицы задаются какой-нибудь простой формулой и, следовательно, могут вычисляться по мере необходимости. Например, рассмотрим уравнение Лапласа:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

При его моделировании конечно-разностными методами элементы $A$ обычно равны одному из чисел 0, 1 или −4; причем какое именно значение принимается, легко определить исходя из геометрии разностной сетки. В таком случае элементы можно вовсе не хранить, а генерировать их, когда они понадобятся. Кроме того, часто порядки $n$ столь велики, что было бы невозможно хранить заполненные массивы в памяти компьютера.

Желательно решать такие линейные системы $A \times \mathbf{x} = \mathbf{b}$ методами, которые вообще не меняют матрицу $A$ и требуют хранения лишь нескольких векторов длины $n$. (Заметим, что вектор **b** обычно должен храниться так же, как и вектор **x**.)

Методы, отвечающие этим требованиям, существуют и называются итерационными. В них, начиная с какого-нибудь начального приближения $\mathbf{x}^{(0)}$, выполняется некоторый вычислительный процесс для определения нового вектора $\mathbf{x}^{(1)}$, используя значения $A$, **b** и $\mathbf{x}^{(0)}$. Затем процесс повторяется для определения $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, $\mathbf{x}^{(4)}$ и т. д. При соответствующих предположениях векторы $\mathbf{x}^{(k)}$ сходятся к пределу при $k \longrightarrow \infty$. Существует множество подобных итерационных процессов. Наиболее успешные из них обязаны своим успехом тесной связи с решаемой задачей. Поэтому редко можно встретить библиотечные подпрограммы итерационных методов.

Хотя итерационный процесс математически может быть несложен, структура матрицы $A$, скорее всего, сложна и специальным образом связана с данной конкретной задачей.

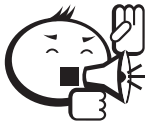
## Метод простой итерации

Самым простым является *метод простой итерации*, который в векторном виде можно представить следующим образом:

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{d},$$

а в координатной форме:

$$\begin{aligned} x_1^{k+1} &= b_{11}x_1^k + b_{12}x_2^k + \ldots + b_{1n}x_n^{(k)} + d_1; \\ x_2^{k+1} &= b_{21}x_1^k + b_{22}x_2^k + \ldots + b_{2n}x_n^{(k)} + d_2; \\ &\ldots \\ x_n^{k+1} &= b_{n1}x_1^k + b_{n2}x_2^k + \ldots + b_{nn}x_n^{(k)} + d_n. \end{aligned}$$



Для сходимости метода простой итерации *достаточно*, чтобы

$$\|B\| < 1$$

или, например,

$$\max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |b_{ij}| < 1.$$

Отсюда вывод: преобразование исходной системы $A \times \mathbf{x} = \mathbf{b}$ к итерационному виду $\mathbf{x}^{(k+1)} = B \times \mathbf{x}^{(k)} + \mathbf{d}$ нужно осуществить таким образом, чтобы коэффициенты при неизвестных в правых частях стали существенно меньше единицы. В самом простом случае $B = A - E$, $\mathbf{d} = -\mathbf{b}$. Если $A$ преобразовать таким образом, чтобы на главной диагонали были только единицы, то получится *метод Якоби* (в этом случае на главной диагонали матрицы $B$ будут только нули).

## Метод Зейделя

Отличие *метода Зейделя* от метода простой итерации состоит лишь в том, что при вычислении $(k + 1)$-ого приближения ранее полученные приближения сразу же используются в вычислениях. В координатной форме итерационный процесс Зейделя имеет вид:

$$\begin{cases} x_1^{(k+1)} = b_{11}x_1^{(k)} + b_{12}x_2^{(k)} + \ldots + b_{1n}x_n^{(k)} + d_1; \\ x_2^{(k+1)} = b_{21}x_1^{(k)} + b_{22}x_2^{(k)} + \ldots + b_{2n}x_n^{(k)} + d_2; \\ \qquad\qquad \ldots \\ x_n^{(k+1)} = b_{n1}x_1^{(k)} + b_{n2}x_2^{(k)} + \ldots + b_{nn}x_n^{(k)} + d_n. \end{cases}$$

Укажем один *практический прием* преобразования исходной системы $A \times \mathbf{x} = \mathbf{b}$ в систему вида $\mathbf{x} = B \times \mathbf{x} + \mathbf{d}$ с гарантией сходимости итерационного процесса метода Зейделя.

Умножим левую и правую части исходной системы на транспонированную матрицу $A^T$. Получим систему вида $C \times \mathbf{x} = \mathbf{d}$, где $C = A^T \times A$; $\mathbf{d} = A^T \times \mathbf{b}$. Эта система называется *нормальной* и обладает следующими свойствами:

1) матрица $c$ является симметричной ($c_{ij} = c_{ji}$);
2) все элементы главной диагонали матрицы $c$ положительны: $c_{ii} > 0$.

После этого проведем уже знакомое нам преобразование: делим каждое уравнение нормальной системы на соответствующий диагональный элемент.

Получаем так называемую приведенную систему:

$$x_i = \sum_{j \neq i} a_{ij} x_j + \beta_i, \quad (i = 1 \ldots n), \quad \text{где } \alpha_{ij} = -\frac{c_{ij}}{c_{ii}}; \quad \beta_i = \frac{d_i}{c_{ii}}; \quad (j \neq i)$$

и уже для приведенной системы записываем итерационный процесс Зейделя:

$$x_i^{(k+1)} = -\sum_{j=1}^{i-1} \frac{c_{ij}}{c_{ii}} x_j^{(k+1)} - \sum_{j=i+1}^{n} \frac{c_{ij}}{c_{ii}} x_j^{(k)} + \frac{d_i}{c_{ii}}.$$

$$x^{(0)} = d.$$

## 3.2 Собственные значения и собственные вектора

Еще одной из задач, помимо решения систем линейных алгебраических уравнений, часто возникающих при осуществлении практической деятельности — это вычисление собственных значений матрицы и соответствующих им собственных векторов. Проблема определения собственных чисел и собственных векторов возникает при анализе схем и конструкций, характеризующихся малыми смещениями от положения равновесия, при анализе устойчивости численных схем, в теории механических и электрических колебаний и т. д.

Различают *полную*, когда необходимо найти все значения, и *частичную*, когда необходимо найти часть значений, *проблему собственных значений*. Задачу нахождения собственных значений и собственных векторов часто называют *второй задачей линейной алгебры*.

***Собственными числами*** *действительной квадратной матрицы* $A$ *называют числа* $\lambda$*, в общем случае комплексные, при которых определитель матрицы:*

$$A - \lambda \cdot E = \begin{pmatrix} a_{11} - \lambda & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} - \lambda & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} - \lambda \end{pmatrix} \tag{3.9}$$

*равен нулю.*

Иными словами, собственное число $\lambda$ матрицы $A$ должно удовлетворять уравнению:

$$|A - \lambda \cdot E| = 0. \tag{3.10}$$

### 3.2.1 Метод непосредственного развертывания

Полную проблему собственных значений для матриц невысокого порядка $(n \leqslant 10)$ можно решить методом непосредственного развертывания.

Если раскрыть определитель из (3.10), то он превратится в полином степени $n$ относительно $\lambda$:

$$P(\lambda) = \lambda^n + p_{n-1} \cdot \lambda^{n-1} + p_{n-2} \cdot \lambda^{n-2} + \ldots + p_1 \cdot \lambda + p_0, \tag{3.11}$$

где $n$ — размер матрицы $A$, а коэффициенты $p_k$ зависят только от значений элементов матрицы $A$. Уравнение (3.10) примет вид:

$$P(\lambda) = \lambda^n + p_{n-1} \cdot \lambda^{n-1} + p_{n-2} \cdot \lambda^{n-2} + \ldots + p_1 \cdot \lambda + p_0 = 0, \tag{3.12}$$

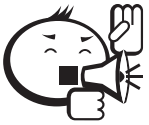
Данное уравнение еще называется характеристическим уравнением. Таким образом, задача о поиске собственных чисел матрицы размера $n \times n$ сводится к поиску корней полинома степени $n$.

В общем случае полином (3.11) может быть представлен в виде произведения:

$$P(\lambda) = (\lambda - \lambda_1)^{m_1}(\lambda - \lambda_2)^{m_2}\ldots(\lambda - \lambda_K)^{m_K}, \tag{3.13}$$

где $\lambda_i$ — $i$-ый корень полинома; $m_i$ — кратность корня $\lambda_i$; $K$ — число различных корней.

В математике есть т. н. *основная теорема алгебры*, которая утверждает: *всякий полином степени* ***n*** *имеет в поле комплексных чисел ровно* ***n*** *корней, причем каждый корень считается столько раз, какова его кратность*. Это означает, что $m_1 + m_2 + \ldots + m_K = n$.



Задача поиска корней полинома в аналитическом виде решена лишь для $n \leqslant 4$, для $n > 4$ возможен только их численный поиск.

С собственным числом матрицы связано понятие *«собственный вектор»*. Собственным вектором матрицы $A$, соответствующим собственному числу $\lambda_i$, называют вектор $t_i$, для которого справедливо соотношение:

$$A \cdot t_i = \lambda_i \cdot t_i, \text{ где } i = 1\ldots n. \tag{3.14}$$

Допустим, что матрица $A$ имеет $n$ различных собственных чисел и соответственно $n$ собственных векторов. Составим матрицу $T$, столбцы которой образованы векторами $t_i$:

$$T = \begin{pmatrix} t_1 & t_2 & \ldots & t_n \end{pmatrix},$$

и запишем уравнения (3.14) в матричной форме:

$$A \cdot T = T \cdot \begin{pmatrix} \lambda_1 & 0 & \ldots & \ldots \\ 0 & \lambda_2 & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & \lambda_n \end{pmatrix}. \tag{3.15}$$

Соотношения (3.14) и (3.15) полностью эквивалентны.

При определении собственных векторов, для каждого собственного числа (корня характеристического уравнения (3.12)) необходимо составить систему:

$$(A - \lambda_i \times E) \times X_i, \; i = 1, 2, \ldots, n$$

и найти собственные векторы $X_i$.

## 3.2.2 Метод итераций

Для решения частичной проблемы собственных значений и собственных векторов на практике часто применяют метод итераций. С его помощью можно приближенно (с заданной точностью $\varepsilon$) получить собственные значения матрицы $A$, имеющей $n$ линейно независимых собственных векторов $X_i$, $1 \leqslant i \leqslant n$ [8].

Алгоритм метода итераций состоит из следующих шагов:

На первом шаге задается начальное приближение (отличное от 0) собственного вектора $X_i^0$, здесь и далее верхний индекс соответствует номеру приближения, а нижний — номеру собственного значения. Шаг итерации $k$ полагаем равным 0.

На втором шаге вычисляем $X_i^1 = A \times X_i^0$, $\lambda_i^1 = x_{j(i)}^1 / x_{j(i)}^0$, где $x_{j(i)}^k$ — $j$-ая координата вектора $X_i^k$, $1 \leqslant j \leqslant n$, причем $j$ может быть любой. Шаг итерации $k$ полагаем равным 1.

Шаг 3. Вычисляем $X_i^{k+1} = A \times X_i^k$.

Шаг 4. Вычисляем $\lambda_i^{k+1} = x_{j(i)}^{k+1} / x_{j(i)}^k$.

Шаг 5. вычисляем $\Delta = |\lambda_i^{k+1} - \lambda_i^k|$. Если $\Delta \leqslant \varepsilon$, вычисления прекращаем и принимаем $\lambda_i \cong \lambda_i^{k+1}$, в противном случае полагаем $k = k + 1$ и переходим к шагу 3.

Процесс приближений $X_i^k = A \times X_i^{k-1} = A \times A^{k-1} \times X_i^0$ сходится при $k \longrightarrow \infty$, и $X_i^k$ стремится к собственному вектору $X_i$.

## 3.3 Интерполяция

Достаточно часто в реальной практике приходится сталкиваться с данными, полученными эмпирически, в результате каких либо наблюдений. Как правило, наблюдения осуществляются в дискретные интервалы времени. Нам доступны только те значения, которые мы наблюдали в эти моменты, а что происходило в промежутке между моментами регистрации данных, нам не известно, а зачастую эти данные необходимы.

Предположим, что задано множество вещественных абсцисс $x_1, \ldots, x_n$ $(x_1 < x_2 < \ldots < x_n)$ и им соответствующие ординаты $y_1, \ldots, y_n$. Задача одномерной интерполяции состоит в построении функции $f$, такой, что $f(x_i) = y_i$ для всех $i$, и при этом $f(x)$ должна принимать «разумные» значения для $x$, лежащих между заданными точками. Критерий разумности слабо формализован, существенно зависит от задачи, и ему невозможно дать точное определение.

Интерполяция часто встречается как при работе на компьютере, так и в обычной жизни. Например, если стрелка спидометра автомобиля (или часов, или любого другого прибора) находится между делениями, мы мысленно интерполируем, чтобы получить скорость. Если у нас есть данные, полученные с большими затратами всего в нескольких точках (например, результаты переписи населения, проводимой раз в десять лет), то может потребоваться определить величины между этими точками. Если ординаты $\{y_i\}$ происходят от гладкой математической функции и ошибки в них не превосходят уровня округлений, то можно рассчитывать, что задача имеет удовлетворительное решение [3].

Если точки $(x_i, y_i)$ получены из очень точных экспериментальных наблюдений, то зачастую их можно считать лишенными ошибок, и тогда вполне разумно интерполировать их гладкой функцией. Если, с другой стороны, эти точки проистекают из сравнительно грубых экспериментов, то неправомерно требовать от интерполирующей функции точно удовлетворять таким данным. Позволяя значениям $f(x_i)$ отличаться от $y_i$, можно очень хорошо отразить характер изменения данных и даже поправить некоторые из содержащихся в них ошибок.

Цели интерполяции разнообразны, но почти всегда в ее основе — желание иметь быстрый алгоритм вычисления значений $f(x)$ для $x$, не содержащихся в таблице данных $(x_i, y_i)$.

Для задачи интерполирования очень важно определение того, как должна вести себя приемлемая функция между заданными точками. В конце концов эти точки могут быть интерполированы бесконечным множеством различных функций, и нужно иметь некоторый критерий выбора. Обычно критерии формулируются в терминах гладкости и простоты; например, функция $f$ должна быть аналитична и максимальное значение $|f(x)|$ по всему интервалу должно быть насколько возможно мало или $f$ должна быть полиномом наименьшей степени и т. п.

Многие интерполирующие функции генерируются линейными комбинациями элементарных функций. Линейные комбинации одночленов $\{x_k\}$ приводят к полиномам. Линейные комбинации тригонометрических функций $\{\cos kx, \sin kx\}$ ведут к тригонометрическим полиномам. Используются также, хотя и реже, линейные комбинации экспонент $\{\exp(b_k x)\}$ или рациональные функции вида:

$$\frac{a_0 + a_1 x + \ldots + a_m x^m}{b_0 + b_1 x + \ldots + b_n x^n}.$$

Рассмотрим сначала полиномиальную интерполяцию, а затем один вид кусочно-полиномиальной интерполяции, так называемую сплайн-интерполяцию.

## 3.3.1 Полиномиальная интерполяция

Наиболее важным классом интерполирующих функций является множество алгебраических полиномов. Полиномы имеют очевидное достоинство — их значения легко вычислять. Их также легко складывать, умножать, интегрировать или дифференцировать. Еще одно свойство полиномов: если $c$ — константа, а $p(x)$ — полином, то полиномами будут и $p(cx)$, и $p(x+c)$. Кроме того, известно, что *любая* непрерывная функция $f(x)$ на замкнутом интервале может быть хорошо приближена некоторым полиномом $p_n(x)$.

Полином степени $n$ можно записать по степеням $x$:

$$y(x) = a_0 + a_1 x + \ldots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

Поскольку здесь $n + 1$ коэффициентов, для однозначного определения коэффициентов нужно задать $n + 1$ корректно поставленных условий. При интерполяции обычно требуют, чтобы полином проходил через $n+1$ точек $(x_i, y_i)$, $(i = 0, 1, \ldots, n)$, где все $x_i$ различны. Это дает $n + 1$ линейных уравнений для неизвестных коэффициентов $a_j$:

$$y_i = \sum_{j=0}^{n} a_j x_i^j \ \ (i = 0, 1, 2, \ldots, n).$$

Можно показать, что решение этой задачи существует и определяется единственным образом.

После того как мы решили для интерполирования воспользоваться многочленом степени $\leqslant n$, в наших руках еще остается возможность выбора базиса в пространстве таких многочленов. Выше был взят базис из одночленов $1, x, x^2, \ldots, x^n$. Это приводит, как мы видели, к системе линейных уравнений, которая в принципе может быть решена методами параграфе 3.1. Однако во многих случаях такие системы уравнений чрезвычайно плохо обусловлены. Предположим, например, что

абсциссы $\{x_i\}$ распределены приблизительно равномерно на интервале $[0, 1]$. Оказывается, что последовательные степени $1, x, x^2, \ldots$ почти линейно зависимы на интервале $[0, 1]$ отчасти потому, что все они положительны и графики всех идут от $(0, 0)$ к $(1, 1)$. Именно эта близость к линейной зависимости делает решение линейной системы при нормальной рабочей точности крайне трудным делом для порядка $n$, превышающего 10.

Гораздо более удовлетворительный способ вычисления полинома, который интерполирует точки $(x_i, y_i)$, состоит в использовании базиса так называемых Лагранжевых полиномов, ассоциированных с множеством $\{x_i\}$. Это полиномы $\{l_j(x)\}$ ($j = 0, 1, \ldots, n$) степени $n$ вида:

$$l_j(x) = \prod_{i=0,\, i\neq j}^{n} \frac{(x - x_i)}{(x_j - x_i)},$$

такие, что

$$l_j(x) = \begin{cases} 1, & \text{если } i = j; \\ 0 & \text{в противном случае.} \end{cases}$$

Легко видеть, что полином степени $n$

$$l_j(x) = \frac{(x - x_0)(x - x_1)\ldots(x - x_{j-1})(x - x_{j+1})\ldots(x - x_n)}{(x_j - x_0)(x_j - x_1)\ldots(x_j - x_{j-1})(x_j - x_{j+1})\ldots(x_j - x_n)}$$

удовлетворяет этим условиям. При этом $l_j$ определяется единственным образом. Каждый множитель числителя обращает $l_j(x_j)$ в нуль при некотором $i \neq j$. Соответствующие множители знаменателя нормируют полином так, что $l_j(x_j) = 1$. Полином $l_j(x_j)y_j$ принимает значение $y_j$ в точке $x_j$ и равен нулю во всех точках $x_i$ $(i \neq j)$. Таким образом, интерполяционный полином степени $n$, который проходит через $n + 1$ точек $(x_i, y_i)$, выражается формулой

$$y(x) = \sum_{j=0}^{n} l_j(x) y_j.$$

Число арифметических операций (и, следовательно, время выполнения) для этого метода пропорционально $n^2$.

Важно подчеркнуть еще раз, что существует *один и только один* полином степени $\leqslant n$, который интерполирует заданные $n + 1$ точек. В литературе [7] имеется множество различных формул для интерполяционных полиномов, основанных на различном выборе базисов; однако при данном наборе точек все они порождают один и тот же полином. Таким образом, полином, получаемый посредством этого подхода, совпадает с полиномом, найденным путем решения линейных уравнений, при условии, что вычисления проводятся в точной арифметике.

Ошибки округлений, соображения, связанные с памятью и временем, могут повлиять на выбор метода. Главное соображение при выборе — это конкретное применение интерполяционного полинома. Коэффициенты такого полинома нужны редко. Обычно Лагранжева интерполяция или какой-либо аналогичный метод являются лучшим выбором.

Существует много обобщений Лагранжевой интерполяции; наиболее употребительна среди них *Эрмитова интерполяция*. Здесь фиксируются $n$ абсцисс

$\{x_i\}$, $n$ заданных значений $\{y_i\}$ и $n$ заданных угловых коэффициентов $\{y_i'\}$. Задача состоит в том, чтобы найти полином $P(x)$ максимальной степени $2n - 1$, такой, что для $i = 1, 2, \ldots, n$

$$P(x_i) = y_i$$

и

$$P'(x_i) = y_i'.$$

При этом, если все $x_i$ различны, то существует единственное решение, и оно может быть построено способом, вполне аналогичным методу Лагранжа.

Помимо вопросов глобальной сходимости, полиномиальная интерполяция имеет и другие недостатки. Время построения и вычисления интерполяционных полиномов высокой степени может для некоторых приложений оказаться чрезмерным. Полиномы высокой степени могут приводить также к трудным проблемам, связанным с ошибками округлений.

## 3.3.2 Сплайн-интерполяция

Полиномиальная интерполяция является глобальной, т. е. полиномиальная функция должна проходить через все заданные точки. При добавлении данных приходится увеличивать степень полинома, что часто приводит к вычислительным трудностям. В таких случаях лучше прибегнуть к альтернативному подходу — использованию *кусочно-полиномиальных* функций. Если при решении таких задач использовать кусочную интерполяцию более низкого порядка (интерполяция осуществляется по небольшому количеству узловых точек отрезка, а затем эти полиномы объединяются в единую интерполяционную формулу), то при этом в точках стыковки обычно терпит разрыв уже первая производная и дифференцировать полученную интерполяционную функцию нельзя. Для того, чтобы построить гладкую интерполяционную функцию, можно использовать так называемую *сплайн-интерполяцию*.

Кубические сплайн-функции — это сравнительно недавнее математическое изобретение, но они моделируют очень старое механическое устройство. Чертежники издавна пользовались механическими сплайнами, представляющими собой гибкие рейки из какого-нибудь упругого материала. Механический сплайн закрепляют, подвешивая грузила в точках интерполяции, называемых узлами. Сплайн принимает форму, минимизирующую его потенциальную энергию, а в теории балок устанавливается, что эта энергия пропорциональна интегралу по длине дуги от квадрата кривизны сплайна. Далее, элементарная теория балок показывает, что сплайн является кубическим полиномом между каждой соседней парой узлов и что соседние полиномы соединяются непрерывно, так же как и их первые и вторые производные.

Построение кубического сплайна — простой и численно устойчивый процесс.

Пусть на $[a, b]$ задана непрерывная функция $f(x)$. Разобьем этот отрезок на $n$ промежутков точками $x_i$ $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$. Пусть $y_k = f(x_k)$, $k = 0, 1, 2, \ldots, n$, $h_j = x_j - x_{j-1}$; $j = 1, \ldots, n$.

***Кубическим сплайном*** *называется функция* $S(x)$*, удовлетворяющая следующим условиям:*

1) *на каждом отрезке* $[x_{k-1}, x_k]$*,* $k = 1, 2, \ldots, n$*, функция* $S(x)$ *является полиномом 3-ей степени;*
2) *функция* $S(x)$*, а также ее первая и вторая производные непрерывны на* $[a, b]$*;*
3) $S(x_k) = f(x_k)$*,* $k = 0, 1, \ldots, n$*.*

Будем искать кубический сплайн в виде

$$S_k(x) = a_k + b_k(x - x_{k-1}) + c_k(x - x_{k-1})^2 + d_k(x - x_{k-1})^3;\ x_{k-1} \leqslant x \leqslant x_k;\ k = 1\ldots n.$$

Значит, чтобы определить функцию $S(x)$, необходимо определить $4n$ коэффициентов $a_k$, $b_k$, $c_k$, $d_k$.

По определению кубического сплайна можно записать следующую систему уравнений для определения этих коэффициентов.

$$S_k(x_{k-1}) = a_k = f(x_{k-1}); \quad (n \text{ уравнений}) \tag{3.16}$$

$$S_k(x_k) = a_k + b_k h_k + c_k h_k^2 + d_k h_k^3 = f(x_k), \quad (n \text{ уравнений}) \tag{3.17}$$

$$S_k'(x_k) = S_{k+1}'(x_k), \quad (n - 1 \text{ уравнений}) \tag{3.18}$$

$$S_k''(x_k) = S_{k+1}''(x_k), \quad (n - 1 \text{ уравнений}) \tag{3.19}$$

Дважды продифференцировав $S_k$, получим

$$S_k'(x) = b_k + 2c_k(x - x_{k-1}) + 3d_k(x - x_{k-1})^2.$$
$$S_k''(x) = 2c_k + 6d_k(x - x_{k-1}).$$

На основании этого (3.11)–(3.12) можно переписать в виде

$$b_k + 2c_k h_k + 3d_k h_k^2 = b_{k+1},\ k = 1, \ldots, n - 1. \tag{3.20}$$

$$c_k + d_k h_k = c_{k+1},\ k = 1, \ldots, n - 1. \tag{3.21}$$

Для замыкания не хватает еще двух уравнений. Обычно их получают из условий на границах отрезка. Существует несколько способов задания граничных условий — заданы первые производные от $S$ в $a$ и $b$, заданы вторые производные и т. д. Для определенности пусть известны первые производные.

$$f'(a) = f_a; \quad f'(b) = f_b.$$

Тогда недостающие два уравнения:

$$b_1 = f_a. \tag{3.22}$$

$$b_n + 2c_n h_n + 3d_n h_n^2 = f(b). \tag{3.23}$$

Для решения этой системы уравнений обычно применяют следующий способ.

Последовательно исключая $a_k$ (из уравнений (3.16) и (3.17)), $d_k$ (из уравнений (3.21)) и, наконец, $b_k$, получаем систему уравнений для нахождения коэффициентов $c_k$, $k = 1, \ldots, n$. Эта система представляет собой систему $n$ линейных алгебраических уравнений с трехдиагональной матрицей, которая может быть решена методом *прогонки*.

## 3.4 Численное интегрирование

Рассмотрим решение следующей задачи — вычисление определенного интеграла:

$$I = \int_a^b f(x)\,dx.$$

Это одна из фундаментальных задач математического анализа, тесно связанная, в частности, с решением дифференциальных уравнений [2].

Если нет возможности выразить интеграл в известных элементарных или специальных функциях, то применяется приближенное численное интегрирование. С другой стороны, при решении многих инженерных задач удачно выбранный численный метод может оказаться экономичней вычисления точного значения интеграла, выраженного через тригонометрические и прочие специальные функции.

Рассмотрим квадратурные формулы (методы численного интегрирования), основанные на интерполировании по небольшому числу точек. Заменяя подынтегральную функцию, например интерполяционным полиномом Лагранжа, получаем приближенную формулу:

$$I = \int_a^b f(x)\,dx \approx \int_a^b L_n(x)\,dx. \tag{3.24}$$

При этом предполагается, что отрезок $[a, b]$ разбит на $n$ частей точками $x_0 = a$, $x_1, x_2, \ldots, x_{n-1}, x_n = b$, по которым строят интерполяционный полином. Подставляя выражение для интерполяционного полинома Лагранжа, получим, что определенный интеграл можно представить в виде:

$$\int_a^b f(x)\,dx \approx \sum_{k=0}^{n} C_k y_k,$$

причем коэффициенты $C_k$ не зависят от подынтегральной функции $f(x)$, а только от значений узлов интерполяции.

Это приближенное равенство называется *квадратурной формулой*. Точки $x_k$ называются *узлами квадратурной формулы*, а числа $C_k$ — *коэффициентами квадратурной формулы*.

Разность

$$R_n(f) = \int_a^b f(x)\,dx - \sum_{k=0}^{n} C_k y_k$$

называется *погрешностью квадратурной формулы*.

Отметим, что:

1) погрешность зависит как от расположения узлов, так и от выбора коэффициентов;
2) если функция $f(x)$ — полином степени $n$ , то тогда квадратурная формула будет точной (то есть $R_n(f) \equiv 0$), так как в этом случае $L_n(x) \equiv f(x)$;

Получим некоторые простые (и в то же время весьма распространенные) формулы численного интегрирования, используя интерполяционный полином Лагранжа.

Для получения формул численного интегрирования на некотором отрезке $[a, b]$ достаточно построить квадратурную формулу для интеграла на элементарном отрезке $[x_i, x_{i+1}]$, а затем ее просуммировать, т. к.

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx.$$

## 3.4.1 Формула прямоугольников

Аппроксимируем подынтегральную функцию (т. е. приближенно заменяя ее) полиномом нулевой степени, т. е. константой $f(x_*)$, где $x_*$ — точка, принадлежащая отрезку $[x_i, x_{i+1}]$. Тогда

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \int_{x_i}^{x_{i+1}} f(x_*)\,dx = (x_{i+1} - x_i) f(x_*).$$

Если, в качестве $x_*$ выбрать середину отрезка: $x_* = (x_i + x_{i+1})/2 = x_{i+1/2}$, то получим формулу «центральных» прямоугольников:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx y_{i+\frac{1}{2}} h_i, \text{ где } h_i = x_{i+1} - x_i;\ y_{i+\frac{1}{2}} = f\left(x_{i+\frac{1}{2}}\right).$$

Просуммировав, получаем *составную (обобщенную) формулу «центральных» прямоугольников*:

$$\int_a^{b_i} f(x)\,dx \approx \sum_{i=0}^{n-1} y_{i+\frac{1}{2}} h_i.$$

Для равномерной сетки $(h = (b - a)/n)$ формула примет вид:

$$\int_a^{b_i} f(x)\,dx \approx h \sum_{i=0}^{n} y_{i+\frac{1}{2}}.$$

В дальнейшем будем рассматривать только равномерную сетку.

Используя разложения в ряд Тейлора, можно показать, что для составной формулы «центральных» прямоугольников ошибку $R$ можно оценить при помощи формулы:

$$R \leqslant M_2 \frac{b-a}{24} h^2,$$

где $M_2 = \max\limits_{x \in [a,b]} |f''(x)|$.

Таким образом, погрешность формулы «центральных» прямоугольников на всем отрезке интегрирования есть величина второго порядка малости по отношению к шагу. Говорят, что квадратурная формула имеет *второй порядок точности*.

Если в качестве узловой точки $x_*$ выбирать $x_i$ или $x_{i+1}$, то получим соответственно квадратурные формулы «левых» прямоугольников:

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{n-1} f(x_i)$$

и «правых» прямоугольников:

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{n-1} f(x_{i+1}).$$

Однако нетрудно показать, что обе эти формулы имеют только первый порядок точности.

## 3.4.2 Формула трапеций

Аппроксимируем подынтегральную функцию полиномом первой степени, построенным по двум узлам: $x_i$ и $x_{i+1}$. Интерполяционный полином Лагранжа первой степени имеет вид:

$$L_1(x) = \frac{x-x_i}{x_{i-1}-x_i} f(x_{i-1}) + \frac{x-x_{i-1}}{x_i - x_{i-1}} f(x_i) = \frac{1}{h}\big((x-x_{i-1})y_i - (x-x_i)y_{i-1}\big).$$

Подставляя это выражение для $L_1(x)$ в (3.16) для $[x_i, x_{i+1}]$, получим:

$$\int_{x_{i-1}}^{x_i} f(x)\,dx \approx \frac{1}{h} y_i \int_{x_{i-1}}^{x_i} (x - x_{i-1})\,dx - \frac{1}{h} y_{i-1} \int_{x_{i-1}}^{x_i} (x - x_i)\,dx = \frac{h}{2}(y_{i-1} + y_i).$$

Эта формула называется *формулой трапеций* для элементарного отрезка $[x_{i-1}, x_i]$. Просуммировав по всему отрезку, получим *составную формулу трапеций*:

$$\int_a^b f(x)\,dx \approx \frac{h}{2} \sum_{i=1}^{n} (y_{i-1} + y_i).$$

Более просто формулу трапеций можно получить, заменив подынтегральную функцию ломанной, последовательно проходящей через точки $\big(x_0, f(x_0)\big)$, $\big(x_1, f(x_1)\big), \ldots, \big(x_{n-1}, f(x_{n-1})\big), \big(x_n, f(x_n)\big)$. Таким образом, интеграл по каждому

элементарному отрезку (значение которого равно площади соответствующей криволинейной трапеции) заменяется площадью соответствующей прямоугольной трапеции.

Погрешность формулы трапеции оценивается выражением:

$$R \leqslant \frac{M_2(b-a)}{12}h^2.$$

Формула трапеций имеет тот же порядок точности — второй, что и формула центральных прямоугольников, но ее погрешность оценивается величиной в два раза большей. Поэтому предпочтительнее пользоваться формулой прямоугольников.

## 3.4.3 Формула Симпсона

Аппроксимируем подынтегральную функцию полиномом второй степени, построенным по трем узлам: $x_{i-1}$, $x_{i-1/2}$, $x_i$. Интерполяционный полином Лагранжа второй степени имеет вид:

$$L_2(x) = \frac{(x-x_{i-\frac{1}{2}})(x-x_i)}{(x_{i-1}-x_{i-\frac{1}{2}})(x_{i-1}-x_i)}f(x_{i-1}) + \frac{(x-x_{i-1})(x-x_i)}{(x_{i-\frac{1}{2}}-x_{i-1})(x_{i-\frac{1}{2}}-x_i)}f(x_{i-\frac{1}{2}}) +$$
$$+ \frac{(x-x_{i-\frac{1}{2}})(x-x_{i-1})}{(x_i-x_{i-\frac{1}{2}})(x_i-x_{i-1})}f(x_i) = \frac{2}{h^2}\Big[(x-x_{i-\frac{1}{2}})(x-x_i)y_{i-1} -$$
$$- 2(x-x_{i-1})(x-x_i)y_{i-\frac{1}{2}} + (x-x_{i-1})(x-x_{i-\frac{1}{2}})y_i\Big].$$

Подставляя это выражение для $L_2(x)$ в (3.16) для $[x_i, x_{i+1}]$, получим:

$$\int_{x_{i-1}}^{x} f(x)\,dx \approx \frac{2}{h^2}y_{i-1}\int_{x_{i-1}}^{x_i}(x-x_{i-\frac{1}{2}})(x-x_i)\,dx - \frac{4}{h^2}y_{i-\frac{1}{2}}\int_{x_{i-1}}^{x_i}(x-x_{i-1})(x-x_i)\,dx +$$
$$+ \frac{2}{h^2}y_i\int_{x_{i-1}}^{x_i}(x-x_{i-1})(x-x_{i-\frac{1}{2}})\,dx = \frac{h}{6}(y_{i-1}+4y_{i-\frac{1}{2}}+y_i).$$

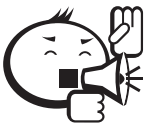
Получили *формулу Симпсона* для элементарного отрезка $[x_{i-1}, x_i]$. Просуммировав по отрезку $[a, b]$, получаем *составную формулу Симпсона*:

$$\int_a^b f(x)\,dx \approx \frac{h}{6}\sum_{i=1}^{n}(y_{i-1}+4y_{i-\frac{1}{2}}+y_i) = \frac{h}{6}\left[y_0+y_n+2\sum_{i=1}^{n-1}y_i+4\sum_{i=1}^{n}y_{i-\frac{1}{2}}\right].$$

Погрешность формулы Симпсона оценивается формулой

$$R \leqslant \frac{M_4(b-a)}{2880}h^4,$$

где $M_4 = \max\limits_{x\in[a,b]}|f^{IV}(x)|$.



Таким образом, формула Симпсона существенно точнее, чем формула прямоугольников или трапеций. Порядок точности — четвертый.

Формула Симпсона называется также формулой парабол, т. к. графиком интерполяционного полинома второго порядка как раз и является парабола.

## 3.5 Численное решение задачи Коши для обыкновенных дифференциальных уравнений

Обыкновенное дифференциальное уравнение (ОДУ) первого порядка можно записать в виде

$$y' = f(y, t). \tag{3.25}$$

Это уравнение имеет семейство решений $y(t)$. Например, если $f(y, t) = y$, то для произвольной константы $C$ функция $y(t) = Ce^t$ является решением. Выбор начального значения, скажем $y(0)$, служит для выделения одной из кривых семейства. Начальное значение зависимой переменной может быть задано для любого значения $t_0$ независимой переменной. Однако часто считают, что выполнено преобразование, обеспечивающее, чтобы $t_0 = 0$. Это не влияет на решение или методы, используемые для приближения решения.

Зачастую имеется более чем одна зависимая переменная, и тогда задача заключается в решении системы уравнений первого порядка; например,

$$\frac{dy}{dt} = f(t, y, z),$$
$$\frac{dz}{dt} = g(t, y, z).$$

Решение этой системы содержит две постоянные интегрирования, и, следовательно, нужны два дополнительных условия, чтобы определить эти константы. Если значения $y$ и $z$ указаны при одном и том же значении независимой переменной $t_0$, то система будет иметь единственное решение. Задача определения значений $y$ и $z$ для (будущих) значений $t > t_0$ называется *задачей Коши*.

Любое обыкновенное дифференциальное уравнение порядка $n$, которое можно записать так, что его левая часть есть производная наивысшего порядка, а в правой части эта производная не встречается, может быть записано и в виде системы из $n$ уравнений первого порядка путем введения $n - 1$ новых переменных. Например, уравнение второго порядка:

$$u'' = g(u, u', t)$$

можно записать как систему:

$$v' = g(u, v, t),$$
$$u' = v.$$

В векторных обозначениях это выглядит так:

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}, t),$$

где:

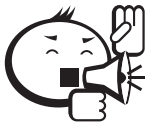
$$\mathbf{y} = \begin{pmatrix} v \\ u \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix};$$

$$\mathbf{f}(\mathbf{y}, t) = \begin{pmatrix} g(y_2, y_1, t) \\ y_1 \end{pmatrix}.$$

При обсуждении методов для задачи Коши удобно представлять себе единственное уравнение:

$$y'(t) = f(y, t);$$
$$y(t_0) = y_0.$$

Однако методы с равным успехом применимы и к системам уравнений.



Лишь очень немногие дифференциальные уравнения могут быть решены точными или приближенными аналитическими методами.

Поэтому (особенно после появления компьютеров) широкое распространение получили *численные методы*, в основе которых лежит замена исходного уравнения его дискретным *аналогом — разностным* уравнением.

Область непрерывного изменения аргумента заменяется дискретным множеством точек (узлов): $t_0$, $t_1$, $t_2$, $t_3$, …, возможно, с переменной длиной шага $h_n = t_{n+1} - t_n$. Эти узлы составляют *разностную сетку*. Искомая функция непрерывного аргумента приближенно заменяется функцией дискретного аргумента на заданной сетке, т. е. в каждой точке $t_n$ решение $y(t_n)$ заменяется приближенным значением (аппроксимируется) $y_n$, которое вычисляется по предыдущим значениям. Эта функция *называется сеточной функцией*. Она определена только в узлах разностной сетки.

Разностный метод, дающий формулу для вычисления $y_{n+1}$ по $k$ предыдущим значениям $y_n$, $y_{n-1}$, $y_{n-2}$, …, $y_{n-k+1}$, называется $k$-шаговым методом. Если $k = 1$, то это одношаговый метод, а при $k > 1$ это многошаговый метод.

## 3.5.1 Метод Эйлера

Примером одношагового метода является метод Эйлера. В методе Эйлера значение $y_{n+1}$ вычисляется посредством прямолинейной экстраполяции из предыдущей точки $y_n$. Рассмотрим задачу Коши:

$$\frac{dy}{dt} = f(y, t);$$
$$y(t_0) = y_0.$$

Наклон решения $y(t)$ в начальной точке можно вычислить по формуле $y_0' = f(y_0, t_0)$. Тогда приближение $y_1$ к $y(t_1)$ можно найти, беря два первых члена ряда Тейлора:

$$y(t_1) \approx y_1 = y_0 + h_0 f(y_0, t_0).$$

Полагая затем $t_2 = t_1 + h_1$, находим:

$$y(t_2) \approx y_2 = y_1 + h_1 f(y_1, t_1)$$

и т. д. В общем случае:

$$y(t_{n+1}) \approx y_n + h_n f(y_n, t_n). \tag{3.26}$$

При этом обычно на каждом новом шаге приближенное решение переходит на другой член семейства решений.

Для некоторых дифференциальных уравнений это явление может привести к большим ошибкам.

Например, при решении уравнения $y' = y$ методом Эйлера ошибки, сделанные на ранних этапах, умножаются с ростом времени на множитель $e^t$. Это явление называется *неустойчивостью* дифференциального уравнения. Иногда можно обойти эту трудность, решая задачу с обращенным временем. Однако при решении системы уравнений факт неустойчивости часто не зависит от направления решения [2].

С другой стороны, для уравнения $y' = -y$ начальные ошибки убывают с ростом $t$. В этом случае говорят об устойчивости дифференциального уравнения. Вообще для дифференциального уравнения $y' = \lambda y$, где $\lambda$ задано, начальные ошибки в решении умножаются при возрастающем $t$ на $e^{\lambda t}$. Если $\lambda < 0$, то начальные ошибки не возрастают, так что уравнение устойчиво. Если $\lambda > 0$, уравнение неустойчиво. Как мы увидим впоследствии, устойчивое дифференциальное уравнение не обязательно легко решается численными методами. Если нужна малая относительная ошибка, то устойчивость дифференциального уравнения может не иметь значения.

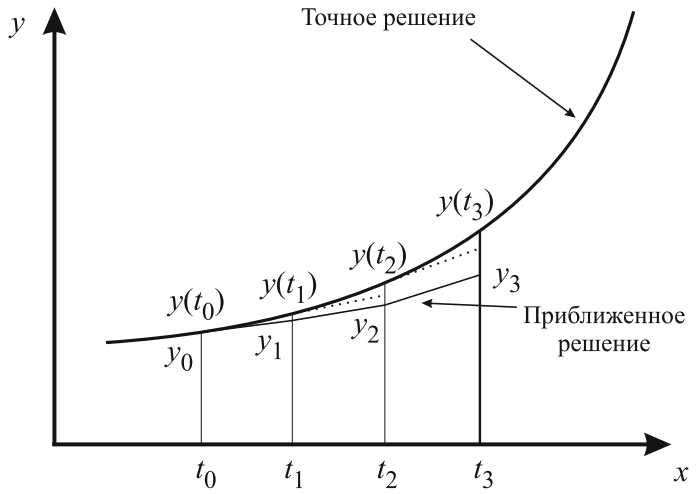



Рис. 3.1 – Геометрическая интерпретация схемы Эйлера

Геометрический смысл схемы Эйлера заключается в замене функции $y(t)$ на отрезке $[t_k, t_{k+1}]$ отрезком касательной, проведенной к графику в точке $t_k$ (рис. 3.1).

## 3.5.2 Ошибки численного интегрирования

Ошибки в численном решении задачи Коши проистекают из двух источников:

1) Ошибка дискретизации.
2) Ошибка округления.

***Ошибка дискретизации*** *есть свойство используемого метода. Это значит, что если бы все арифметические вычисления могли выполняться с бесконечной точностью, то других ошибок, кроме ошибки дискретизации, не было бы.*

***Ошибка округления*** *есть свойство компьютера и программы.*

Поскольку точное решение, вообще говоря, неизвестно и не может быть вычислено, то ошибка дискретизации должна, так или иначе, приближенно оцениваться.

Важно различать между собой локальную и глобальную ошибки дискретизации. Локальная ошибка дискретизации — это ошибка, сделанная на данном шаге при условии, что предыдущие значения точны и что нет ошибок округления. Более точно, локальная ошибка дискретизации $d_n$ есть разность между вычисленным решением и теоретическим решением, определяемыми одними и теми же данными в точке $t_n$. Глобальная ошибка дискретизации есть разность между вычисленным решением (ошибки округления игнорируются) и точным решением, определяемым исходным условием в точке $t_0$.

Различие между локальной и глобальной ошибками дискретизации легко видеть в специальном случае, когда $f(y, t)$ не зависит от $y$. В этом случае решение есть просто интеграл:

$$y(t) = \int_{t_0}^{t} f(\tau)\, d\tau.$$

В этом случае метод Эйлера превращается в схему численного интегрирования по формуле левых прямоугольников. Локальная ошибка дискретизации есть ошибка на одном подинтервале, а глобальная ошибка дискретизации — общая ошибка. В данном специальном случае каждый подинтервал независим от других (сумма может вычисляться в любом порядке), так что глобальная ошибка есть сумма локальных ошибок.

В случае дифференциального уравнения, где $f(y, t)$ зависит от $y$, ошибка на любом интервале зависит от решений, вычисленных для предыдущих интервалов. Вследствие этого глобальная ошибка в общем случае будет больше суммы

локальных ошибок, если дифференциальное уравнение неустойчиво, но меньше этой суммы, если дифференциальное уравнение устойчиво.

Устойчивость дифференциального уравнения и, следовательно, влияние локальной ошибки на глобальную управляются знаком производной $\partial f/\partial y$. Для нелинейных уравнений $\partial f/\partial y$ может изменять знак, так что уравнение может быть неустойчиво в одних областях и устойчиво в других. Для системы нелинейных уравнений ситуация еще более сложна.

Фундаментальным понятием при оценке точности численного метода является его порядок. Порядок определяется в терминах локальной ошибки дискретизации, получаемой при применении метода к задачам с гладкими решениями. Говорят, что метод имеет порядок $p$, если существует число $C$, такое, что:

$$|d_n| \leqslant Ch_n^{p+1}.$$

Обычно число $C$ зависит от производных функции, определяющих дифференциальное уравнение, и от длины интервала, на котором ищется решение, но оно не должно зависеть от номера шага $n$ и величины шага $h_n$. Например, метод Эйлера имеет первый порядок.

Рассмотрим теперь *глобальную* ошибку дискретизации в фиксированной конечной точке $t = t_j$. По мере повышения требований к точности, длины шагов $h_n$ будут убывать, а общее их число $N$, необходимое для достижения $t_j$, будет возрастать. Грубо говоря:

$$N = \frac{t_j - t_0}{h},$$

где $h$ — средняя длина шага. Далее, глобальная ошибка $e_N$ может быть представлена как сумма $N$ локальных ошибок с множителями, описывающими устойчивость уравнения. Эти множители лишь слабо зависят от величин шагов, и потому мы можем, огрубляя, сказать, что если локальная ошибка есть $O(h^{p+1})$, то глобальная ошибка будет $N\,O(h^{p+1}) = O(h^p)$. Вот почему в определении порядка в показателе было взято $p + 1$, а не $p$.

Для метода Эйлера $p = 1$, так что уменьшение средней длины шага в 2 раза уменьшит среднюю локальную ошибку примерно в $2^{p+1} = 4$ раза; но так как для достижения $t_j$ теперь потребуется приблизительно вдвое больше шагов, то глобальная ошибка уменьшится лишь в $2^p = 2$ раза. Для методов более высокого порядка глобальная ошибка для гладких решений уменьшилась бы гораздо ощутимей.

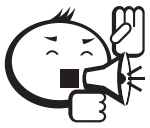
На практике проведенный анализ должен быть дополнен, чтобы учесть ошибки округлений. Пусть на каждом шаге метода Эйлера делается наихудшая возможная ошибка округления $\varepsilon$. Можно надеяться, однако, что вместо того, чтобы быть константой, $\varepsilon$ является в известной степени случайной величиной со средним значением 0. Если при выполнении операций с плавающей точкой компьютер результат округляет, то на практике общая ошибка, накопившаяся от округлений, обычно ведет себя как $\varepsilon\sqrt{N}$. На компьютерах, которые после арифметических операций просто отбрасывают лишние разряды, ошибка округлений имеет тенденцию расти линейно по $N$.

Суммируя глобальную ошибку дискретизации и ошибку округлений, получаем:

$$\textit{общая ошибка} \approx b\left(Ch + \frac{\varepsilon}{h}\right).$$

Следовательно, общая ошибка имеет минимальное значение для некоторого оптимального значения $h$, выражаемого приближенной формулой:

$$h_{opt} \approx \sqrt{\frac{\varepsilon}{C}}.$$



По мере убывания $h$ ошибка дискретизации убывает (линейно), и численное решение начинает сходиться к точному решению. Однако, когда $h$ становится слишком мало, численное решение начинает расходиться вследствие ошибок округлений.

### 3.5.3 Методы Рунге — Кутта

Метод Эйлера является частным случаем широко распространенных в вычислительной практике семейства методов, известных под названием «методы Рунге — Кутта».

Эти методы обладают следующими отличительными свойствами:

1) они являются *одношаговыми*, т. е. чтобы найти $y_{k+1}$, нужна информация только о предыдущей точке: $x_k$, $y_k$, и *явными* (явная зависимость $y_{k+1}$ от $y_k$);
2) они имеют порядок $h^p$, где $p$ — степень порядка (различна для различных методов);
3) они не требуют вычисления производных от $f(x, y)$, а требуют только вычисления самой функции.

Однако для вычисления одной последующей точки решения приходится вычислять функцию $f(x, y)$ несколько раз при различных значениях $x$ и $y$. Это та цена, которую приходится платить за право не вычислять никаких производных, но цена более чем умеренная.

Метод Эйлера — метод Рунге — Кутта первого порядка точности. Его медленная сходимость характерна для всех методов первого порядка и служит препятствием для его широкого использования.

Все методы Рунге — Кутта можно представить в виде:

$$y_{k+1} = y_k + h\varphi(t_k, y_k)$$

с соответствующей функцией $\varphi$. Для метода Эйлера функцией $\varphi$ является сама функция $f$. Рассмотрим частные случаи.

#### Модифицированный метод Эйлера

$$y_{k+1} = y_k + hf\left(t_k + \frac{h}{2}, y_k + \frac{h}{2}f(t_k, y_k)\right). \tag{3.27}$$

*Геометрическая интерпретация*. Через точку A проводится прямая, угловой коэффициент которой равен угловому коэффициенту касательной BC к интегральной кривой $y_B(x)$, проходящей через промежуточную точку B, построенную по методу Эйлера с шагом $h/2$ (рис. 3.2).

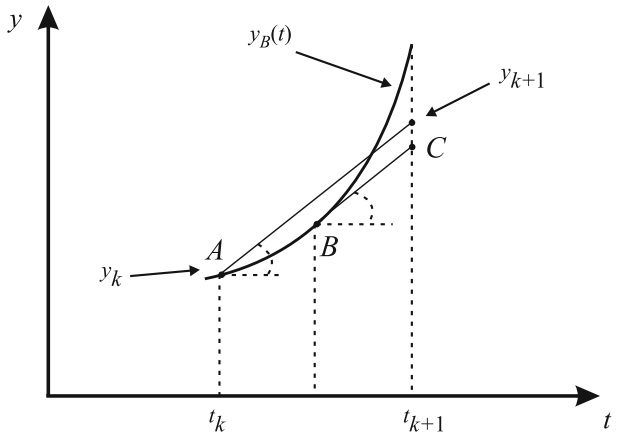



Рис. 3.2 – Геометрическая интерпретация модифицированной схемы Эйлера

## Улучшенный метод Эйлера

$$y_{k+1} = y_k + \frac{h}{2}\left[f(t_k, y_k) + f\left(t_{k+1}, y_k + hf(t_k, y_k)\right)\right]. \tag{3.28}$$

*Геометрическая интерпретация*. Через точку A проводится не касательная к интегральной кривой $y(x)$, а прямая AB' с угловым наклоном, равным среднеарифметическому угловых наклонов касательных AB и BC, которые строятся по методу Эйлера в точках A и B. Решением служит ордината точки B', которая образована пересечением этой прямой с прямой $x = x_{k+1}$ (рис. 3.3).

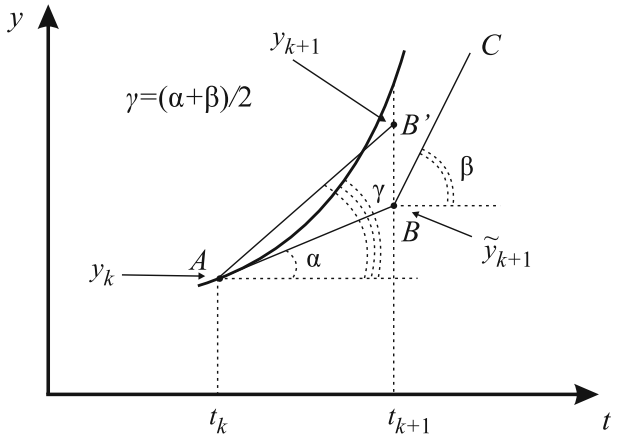



Рис. 3.3 – Геометрическая интерпретация улучшенной схемы Эйлера

Обратите внимание, что эти схемы можно трактовать как схемы «предиктор — корректор» (т. е. «счет — пересчет», «предсказание — уточнение»). Например, схе-

ма (3.3): сначала по схеме Эйлера делается шаг предиктора и находится $\widetilde{y}_{k+1} = y_k + hf(t_k, y_k)$, а затем полусуммой делается шаг корректора:

$$\frac{y_{k+1} - y_k}{h} = \frac{1}{2}\left[f(t_k, y_k) + f(t_{k+1}, \widetilde{y}_{k+1})\right].$$

Наиболее знаменитой схемой метода Рунге — Кутта является классический метод *четвертого порядка*:

$$y_{k+1} = y_k + \frac{1}{6}(m_1 + 2m_2 + 2m_3 + m_4), \tag{3.29}$$

где

$$m_1 = hf(t_k, y_k); \quad m_2 = hf\left(t_k + \frac{h}{2}, y_k + \frac{m_1}{2}\right);$$

$$m_3 = hf\left(t_k + \frac{h}{2}, y_k + \frac{m_2}{2}\right); \quad m_4 = hf\left(t_{k+1}, y_k + m_3\right).$$

Широко используется также модификация классического метода Рунге — Кутта — метод Рунге — Кутта — Мерсона:

$$y_{k+1} = y_k + \frac{1}{6}(m_1 + 4m_4 + m_5); \tag{3.30}$$

$$m_1 = hf(t_k, y_k); \quad m_2 = hf\left(t_k + \frac{h}{3}, y_k + \frac{m_1}{3}\right);$$

$$m_3 = hf\left(t_k + \frac{h}{3}, y_k + \frac{m_1}{6} + \frac{m_2}{6}\right);$$

$$m_4 = hf\left(t_k + \frac{h}{2}, y_k + \frac{m_1}{8} + \frac{3m_3}{8}\right);$$

$$m_5 = hf\left(t_{k+1}, y_k + \frac{m_1}{2} - \frac{3m_3}{2} + 2m_4\right).$$

Методы Рунге — Кутта без труда переносятся на системы обыкновенных дифференциальных уравнений:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y});$$

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_M \end{pmatrix}; \quad \mathbf{f} = \begin{pmatrix} f_1(t, y_1, y_2, \dots, y_M) \\ f_2(t, y_1, y_2, \dots, y_M) \\ \dots \\ f_M(t, y_1, y_2, \dots, y_M) \end{pmatrix}.$$

Запишем, например, формулы метода Рунге — Кутта 4-го порядка для систем двух уравнений:

$$\frac{dy}{dx} = \varphi(t, y, z); \quad \frac{dz}{dx} = \psi(t, y, z).$$

$$y_{k+1} = y_k + \frac{1}{6}(m_1 + 2m_2 + 2m_3 + m_4); \quad z_{k+1} = z_k + \frac{1}{6}(n_1 + 2n_2 + 2n_3 + n_4).$$

$$m_1 = h\varphi(t_k, y_k, z_k); \quad n_1 = h\psi(t_k, y_k, z_k);$$

$$m_2 = h\varphi\left(t_k + \frac{h}{2}, y_k + \frac{m_1}{2}, z_k + \frac{n_1}{2}\right); \quad n_2 = h\psi\left(t_k + \frac{h}{2}, y_k + \frac{m_1}{2}, z_k + \frac{n_1}{2}\right);$$

$$m_3 = h\varphi\left(t_k + \frac{h}{2}, y_k + \frac{m_2}{2}, z_k + \frac{n_2}{2}\right); \quad n_3 = h\psi\left(t_k + \frac{h}{2}, y_k + \frac{m_2}{2}, z_k + \frac{n_2}{2}\right);$$

$$m_4 = h\varphi(t_{k+1}, y_k + m_3, z_k + n_3); \quad n_4 = h\psi(t_{k+1}, y_k + m_3, z_k + n_3).$$

### 3.5.4 Многошаговые методы Адамса

В отличие от одношаговых методов Рунге — Кутта эти методы позволяют найти решение с использованием известных решений в *нескольких* соседних точках.

Итак, предположим, что с помощью какого-нибудь метода уже получена таблица значений:

$$t_0, t_1, t_2, \ldots, t_n,$$
$$y_0, y_1, y_2, \ldots, y_n.$$

Пусть шаг постоянен, т. е. $t_n - t_{n-1} = h$.

Найдем значение в точке $t_{n+1}$. Проинтегрируем исходное уравнение:

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} f\big(t, y(x)\big)\, dt.$$

Аппроксимируя подынтегральную функцию одного переменного $f\big(t, y(t)\big)$ интерполяционным многочленом, который в узлах $t_n$, $t_{n-1}$, $t_{n-2}$, ... принимает соответствующие значения $f\big(t_n, y(t_n)\big)$, $f\big(t_{n-1}, y(t_{n-1})\big)$, получим *экстраполяционную* формулу Адамса в разностном виде:

$$y_{n+1} = y_n + h\left(f_n + \frac{1}{2}\Delta^1 f_{n-1} + \frac{5}{12}\Delta^2 f_{n-2} + \frac{3}{8}\Delta^3 f_{n-3} + \frac{251}{720}\Delta^4 f_{n-4} + \frac{95}{288}\Delta^5 f_{n-5} + \ldots\right) + R_n,$$

где $\Delta^k f_i$ — конечная разность $k$-го порядка $\Delta^k f_i = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i$.

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}\int_0^1 t(t+1)\ldots(t+n)\, dt.$$

Оставляя в этой формуле несколько начальных слагаемых, получают *явные формулы Адамса* для численного интегрирования ОДУ.

1) Если оставить одно слагаемое в сумме, то получили формулу Эйлера:

$$y_{n+1} = y_n + hf_n.$$

Это формула имеет первый порядок. Локальная погрешность:

$$\widetilde{R}_n = \frac{1}{2}h^2 y''(\xi).$$

2) Оставляя два слагаемых в сумме, получим формулу второго порядка:

$$y_{n+1} = y_n + hf_n + \frac{h}{2}\Delta^1 f_{n-1} = y_n + \frac{h}{2}(3f_n - f_{n-1}).$$

Локальная погрешность этой формулы $\widetilde{R}_n = \frac{5}{12}h^3 y^{(3)}(\xi)$.

3) Ограничиваясь тремя слагаемыми в сумме, получим формулу третьего порядка:

$$y_{n+1} = y_n + h\left(f_n + \frac{1}{2}\Delta^1 f_{n-1} + \frac{5}{12}\Delta^2 f_{n-2}\right) = y_n + \frac{h}{12}(23f_n - 16f_{n-1} + 5f_{n-2}).$$

Локальная погрешность этой формулы $\widetilde{R}_n = \frac{3}{8}h^4 y^{(4)}(\xi)$.

Для того, чтобы воспользоваться *формулами Адамса*, необходимо предварительно узнать недостающие начальные значения. Например, в формуле Адамса третьего порядка:

$$y_{k+3} = y_{k+2} + \frac{h}{12}(23f_{k+2} - 16f_{k+1} + 5f_k),\ k = 0, 1, 2, \ldots$$

необходимо знать (при $k = 0$) $(t_0, y_0)$, $(t_1, y_1)$, $(t_2, y_2)$. Эти недостающие начальные значения могут быть найдены с помощью одношаговых методов, например методом Рунге — Кутта соответствующей точности. Но кроме этого недостатка, методы Адамса имеют и некоторые преимущества по сравнению с методом Рунге — Кутта.

Например, в методе Рунге — Кутта четвертого порядка на каждом шаге требуется *четыре раза* вычислить значение правой части. В методе же Адамса четвертого порядка:

$$y_{n+1} = y_n + \frac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})$$

требуется только *один раз* вычислить правую часть (т. е. $f_n$) на каждом шаге, тогда как другие значения были найдены на предыдущих шагах.

## 3.5.5 Неявные разностные формулы

До сих пор мы записывали явные разностные схемы, т. е. необходимое значение функции определялось явным образом по известным точкам.

Для многих классов задач широко используются также неявные схемы. Рассмотрим один из способов их получения.

Аппроксимируем подынтегральную функцию интерполяционным полиномом, начиная с точки $x_{n+1}$, а не с точки $x_n$, как это делалось выше.

Проведя интегрирование, получаем *неявную формулу Адамса* (в разностном виде):

$$y_{n+1} = y_n + h\left(f_{n+1} - \frac{1}{2}\Delta f_n - \frac{1}{12}\Delta^2 f_{n-1} - \frac{1}{24}\Delta^3 f_{n-2} - \frac{19}{720}\Delta^4 f_{n-3} - \frac{3}{160}\Delta^5 f_{n-4} - \ldots\right).$$

Оставляя в этой формуле несколько начальных слагаемых, получают *неявные формулы Адамса* для численного интегрирования ОДУ:

1) $y_{n+1} = y_n + hf_{n+1}$ (неявная формула Эйлера).

Эта формула имеет первый порядок. Ее локальная погрешность $R_n = -\frac{1}{2}h^2 y^{(2)}$.

2) $y_{n+1} = y_n + hf_{n+1} - \frac{h}{2}\Delta f_n = y_n + \frac{h}{2}(f_{n+1} + f_n)$.

Формула имеет второй порядок. Ее локальная погрешность $R_n = -\frac{1}{12}h^3 y^{(3)}$.

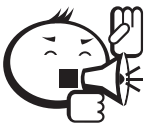
3) $y_{n+1} = y_n + h\left(f_{n+1} - \frac{1}{2}\Delta f_n - \frac{1}{12}\Delta^2 f_{n-1}\right) = y_n + \frac{h}{12}(5f_{n+1} + 8f_n - f_{n-1})$.

Формула имеет второй порядок. Ее локальная погрешность $R_n = -\frac{1}{24}h^4 y^{(4)}$.

4) $y_{n+1} = y_n + h\left(f_{n+1} - \frac{1}{2}\Delta f_n - \frac{1}{12}\Delta^2 f_{n-1} - \frac{1}{24}\Delta^3 f_{n-2}\right) = y_n + \frac{h}{24}(9f_{n+1} + 19f_n - 5f_{n-1} + 5f_{n-2})$.

Формула четвертого порядка. Локальная погрешность $R_n = -\frac{19}{720}h^5 y^{(5)}$.

Основной *недостаток* неявных разностных формул, по сравнению с явными, заключается в том, что для нахождения искомого значения функции $y$ необходимо в общем случае организовывать *итерационный процесс*, используя те или иные формулы приближенного решения трансцендентных уравнений (некоторые из которых будут рассмотрены в параграфе 3.6).



Преимущество неявных разностных формул в следующем:

1) Используют меньше точек, чем соответствующие явные схемы.
2) Их точность выше, чем у соответствующих явных схем.
3) Шире диапазон устойчивости по сравнению с явными схемами.

## 3.5.6 Устойчивость разностных схем

Последнее свойство неявных разностных схем необходимо рассмотреть более подробно. Для каждой конкретной дифференциальной задачи и конкретной разностной схемы существует диапазон устойчивости схемы, который определяется *ограничениями*, накладываемыми на шаг разностной схемы. Необходимо определить, при какой *максимально большой* величине шага решение будет устойчивым.

Проиллюстрируем это на примере. Рассмотрим линейное диффиренциальное уравнение вида:

$$y' = \lambda y, \quad y(0) = y_0,$$

где $\lambda < 0$ — некоторая константа. Точное решение этой задачи $y(x) = e^x y_0$. При возрастании $x$ это решение убывает, сохраняя знак.

1) Для явного метода Эйлера имеем:

$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n.$$

Потребуем, чтобы решение разностной схемы обладало таким же свойством, как и точное решение. Условием такого поведения решения является выполнение неравенства:

$$0 < 1 + h\lambda < 1.$$

Отсюда следует, что

$$h|\lambda| < 1.$$

2) Для явного метода Рунге — Кутта второго порядка решение разностной задачи *монотонно* убывает при выполнении условия

$$|\lambda|h < 2.$$

3) Для явного метода Адамса второго порядка:

$$y_{n+1} = y_n + \frac{h}{2}(3\lambda y_n - \lambda y_{n-1})$$

необходимо выполнение условия

$$|\lambda h| \leqslant 1.$$

4) Для неявного метода Эйлера (и неявного метода Адамса первого порядка):

$$y_{n+1} = y_n + h\lambda y_{n+1},$$

аналогичное условие запишется в виде

$$|1 - \lambda h| > 1,$$

что для $\lambda < 0$ выполняется всегда.

5) Аналогично для неявного метода Адамса второго порядка:

$$y_{n+1} = y_n + \frac{h}{2}(f_{n+1} + f_n);$$

$$\left|\frac{1 + \lambda\frac{h}{2}}{1 - \lambda\frac{h}{2}}\right| < 1,$$

что для всех $\lambda < 0$ выполняется также всегда.

Таким образом, мы видим, что при использовании явных схем существуют довольно *жесткие* ограничения на шаг. Например, для модельной задачи они имеют вид $h \leqslant C/|\lambda|$.

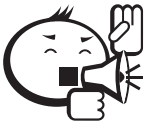
При больших $|\lambda|$ это условие ограничивает выбор шага. Отсюда возник термин «жесткие уравнения». Необходимо отметить, что «жесткость» является свойством уравнения, а не численной схемы. Следовательно, для обеспечения устойчивости явной схемы необходимо выбирать очень маленький шаг интегрирования, что приводит к большому числу шагов на больших промежутках интегрирования и, как следствие, чрезмерному возрастанию *времени* решения задач на ЭВМ и увеличению *вычислительной погрешности*.

## 3.6 Решение трансцендентных уравнений

Пусть $f$ — некоторая функция одной переменной. Задача состоит в том, чтобы найти одно или более решений уравнения $f(x) = 0$. Используемые алгоритмы являются итерационными, и здесь важны такие вопросы: много ли требуется вычислений функции $f(x)$, нужны ли вычисления производных $f'(x)$ и $f''(x)$ и т. д. Чтобы начать поиск нуля $f(x)$, предположим, что можно найти интервал $[a, b]$, на котором $f(x)$ меняет знак. Однако если мы ничего не знаем относительно $f$, то мы не можем быть уверены, что она имеет нуль на этом интервале. Даже если *математическая* функция непрерывна и изменяет знак в $[a, b]$, *вычисляемая* функция переменного с плавающей точкой, значения которой также являются числами с плавающей точкой, принимает лишь дискретное множество значений, среди которых, возможно, нет нуля. Поэтому в конечном счете более практично искать не нуль $f$, а малый интервал $[\alpha, \beta]$, в котором $f$ меняет знак. Такой интервал всегда можно найти и можно сузить его настолько, насколько позволяет система чисел с плавающей точкой, т. е. так, чтобы концевыми точками были два соседних числа этой системы. Если о функции $f$ ничего не известно, то наиболее надежным алгоритмом является метод бисекции или половинного деления.

Метод половинного деления, при заданной точности $\varepsilon$, состоит из таких шагов:

1) Положить $\alpha = a$ и $\beta = b$. Вычислить $f(\alpha)$ и $f(\beta)$.
2) Положить $\gamma = (\alpha + \beta)/2$. Вычислить $f(\gamma)$.
3) Если $\text{sign}(f(\gamma)) = \text{sign}(f(\alpha))$, то заменить $\alpha$ на $\gamma$ (т. е. перенести левую границу отрезка в его середину); в противном случае заменить $\beta$ на $\gamma$ (т. е. перенести правую границу отрезка в его середину). Сущность данного метода заключается в сохранении в любом случае разных знаков у значения функции $f$ на границах отрезка.
4) Если $\beta - \alpha > \varepsilon$, то перейти к шагу 2; в противном случае закончить.



Алгоритм бисекции довольно медленен, но зато абсолютно застрахован от неудачи.

Если каждое вычисление $f(x)$ несложно, то обычно нет серьезных причин, чтобы отвергнуть этот метод. Однако добавочная скорость очень полезна, если вычисление $f(x)$ требует много времени.

Если математическая функция $f$ достаточно гладкая (имеет одну или две непрерывные производные), то часто есть возможность значительно сократить число вычислений функции по сравнению с методом бисекции. Существует большое число различных итерационных методов, из которых мы рассмотрим метод Ньютона и метод секущих.

В **методе Ньютона** нуль $r$ функции $f$ находится как предел последовательности действительных чисел $\{x_k\}$. (Мы предполагаем сейчас точную арифметику действительных чисел.) Каждое новое приближение $x_{k+1}$ вычисляется как единствен-

ный нуль касательной прямой к функции $y = f(x)$ в точке $x_k$, т. е. путем локальной линеаризации $f$ около $x_k$.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

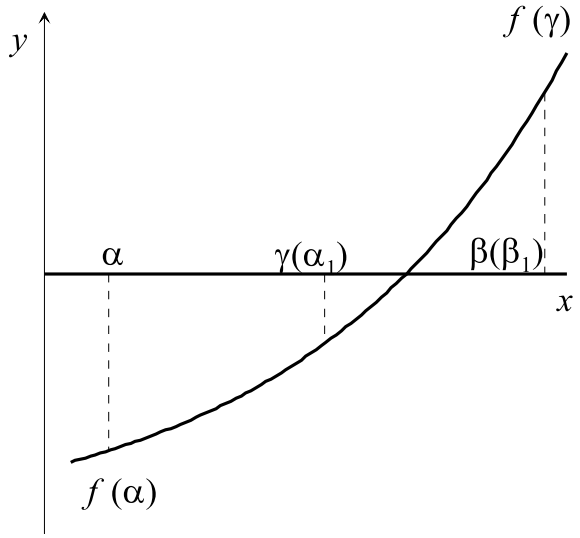



Рис. 3.4 – Геометрическая интерпретация метода половинного деления

Геометрическая интерпретация этого метода заключается в замене на каждой итерации графика $y = f(x)$ касательной к нему в этой точке.

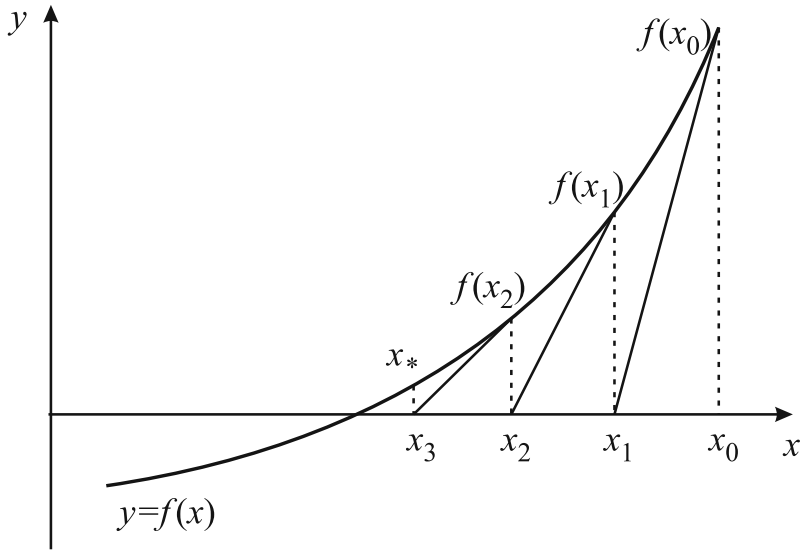



Рис. 3.5 – Геометрическая интерпретация метода Ньютона

Если начальное приближение выбрано *достаточно близко* к корню, то Ньютоновские итерации достаточно быстро сходятся. «Хорошим» начальным приближением $x_0$ является то, для которого выполняется неравенство $f(x_0)f''(x_0) > 0$. То есть в качестве исходной точки $x_0$ можно выбрать тот конец интервала $[a, b]$, которому отвечает ордината того же знака, что и знак $f''(x_0)$.

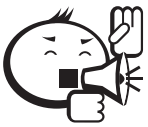
В качестве модификации метода Ньютона можно предложить вычисление производных не на каждом шаге итерационного процесса, а только на первом. Найденное ее значение используют в дальнейшем:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_0)}.$$

Это соответствует замене касательных прямыми параллельными к первой касательной в точке $x = x_0$. Данный прием снижает скорость сходимости, но зато избавляет от необходимости вычисления производных на каждом итерационном шаге.

Методу Ньютона часто предшествует какой-нибудь глобально сходящийся алгоритм типа бисекции, прежде чем можно будет переключиться на быстро сходящиеся итерации методом Ньютона. Таким образом, метод Ньютона зачастую является лишь завершающей процедурой более медленного, но зато гарантированного начального алгоритма. При таком комбинировании, к примеру, последние несколько десятков итераций методом бисекции могут быть заменены несколькими шагами методом Ньютона.

Заметим, что каждая итерация метода Ньютона требует вычисления не только $f(x)$, но и $f'(x)$. Есть функции, для которых вычисление $f'(x)$ после того, как найдено $f(x)$, очень «дешево». Для других функций стоимость вычисления $f'(x)$ эквивалентна второму вычислению $f(x)$. Наконец, для третьих функций вычисление $f'(x)$ почти невозможно.



Главное достоинство метода Ньютона состоит в том, что с его помощью можно находить комплексные нули аналитических функций $f$, а также в том, что его можно распространить на решение систем нелинейных уравнений с многими переменными.

При нахождении нулей функции $f$, для которой вычисление $f'(x)$ затруднено, часто лучшим выбором, чем метод Ньютона, является ***метод секущих***. В этом алгоритме начинают с двумя исходными числами $x_1$ и $x_2$. На каждом шаге $x_{k+1}$ получают из $x_k$ и $x_{k-1}$ как единственный нуль линейной функции, принимающей значения $f(x_k)$ в $x_k$ и $f(x_{k-1})$ в $x_{k-1}$. Эта линейная функция представляет секущую к кривой $y = f(x)$, проходящую через точки с абсциссами $x_k$, $x_{k-1}$ и ординатами $f(x_k)$ , $f(x_{k-1})$ соответственно, — отсюда название: метод секущих (рис. 3.6).

Легко показать что:

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k).$$

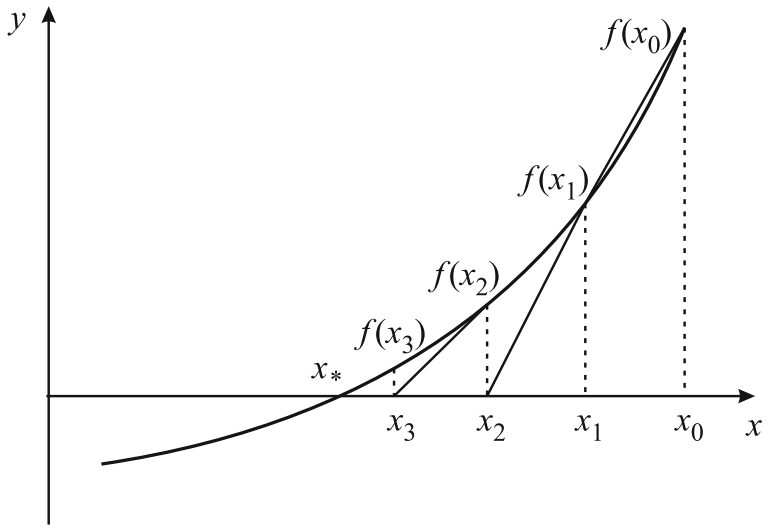



Рис. 3.6 – Геометрическая интерпретация метода секущих

Как и метод Ньютона, метод секущих очень хорошо работает для аналитических функций комплексного переменного. Однако обобщение метода на системы уравнений довольно трудно, хотя и возможно. Подобно методу Ньютона, наибольшая трудность в методе секущих заключается в нахождении $x_1$ и $x_2$, достаточно близких к $r$ для того, чтобы могла начаться сходимость.

## Контрольные вопросы по главе 3

1) Какие вы знаете особенности машинной арифметики?
2) Какие виды матриц различают при численном решении систем линейных уравнений?
3) Что такое ленточные матрицы? Какое их свойство используется при численном решении систем линейных уравнений?
4) В чем заключается метод исключения Гаусса?
5) Что такое обусловленность матрицы? Как можно оценить число обусловленности в процессе численного решения задачи?
6) Какие методы применяются для решения больших разреженных систем линейных уравнений?
7) Сравните метод исключения Гаусса и метод прогонки.
8) Чем отличаются методы Якоби и Зейделя? Как можно обеспечить гарантированную сходимость этих методов?
9) Какие вы знаете проблемы вычисления собственных значений?
10) Сопоставьте по трудности проблемы вычисления собственных значений?

11) К какой задаче сводится метод непосредственного развертывания при вычислении собственных значений.
12) Почему метод непосредственного развертывания неприменим для матриц большой размерности.
13) Сколько собственных векторов можно найти с помощью метода итераций?
14) Насколько точным получается значение собственного вектора, найденного методом итераций.
15) Что такое интерполяция? Какие методы интерполяции существуют?
16) Для чего применяется схема Горнера?
17) Какими недостатками обладает глобальная интерполяция?
18) В каких случаях используется сплайн-интерполяция?
19) Дайте понятие квадратурной формулы. Что такое узлы квадратурной формулы и коэффициенты квадратурной формулы.
20) Что такое погрешность и порядок точности квадратурной формулы?
21) Сравните методы прямоугольников, трапеций и Симпсона.
22) На примере метода Эйлера дайте понятие о численном интегрировании обыкновенных дифференциальных уравнений.
23) Что такое устойчивость дифференциального уравнения?
24) Дайте понятие об ошибках дискретизации и ошибках округления при численном решении ОДУ.
25) Дайте геометрическую интерпретацию модифицированного и улучшенного методов Эйлера.
26) Сравните методы Рунге — Кутта и методы Адамса.
27) Дайте понятие о неявных разностных схемах. В чем их преимущество над явными?
28) Что такое устойчивость разностных схем?
29) Какие методы решения нелинейных уравнений вы знаете?
30) В чем достоинства и недостатки метода Ньютона?
31) Дайте геометрическую интерпретацию метода Ньютона и метода секущих.

# РАЗДЕЛ II

# Прикладной

# Глава 4

# МЕТОДЫ РЕШЕНИЯ СЛАУ

## 4.1 Метод Гаусса

При написании программы, реализующей этот вычислительный процесс, необходимо учитывать одно важное свойство алгоритма. А именно: когда вычисляются коэффициенты $a^{(k+1)}$ и $b^{(k+1)}$, то нужны только коэффициенты $a^{(h)}$ и $b^{(h)}$ при $h = k$ и не нужны коэффициенты с $h < k$,

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{kj}^{k}}{a_{kk}^{(k)}} a_{ik}^{(k)},$$

$$b_{i}^{(k+1)} = b_{i}^{(k)} - \frac{b_{k}^{k}}{a_{kk}^{(k)}} a_{ik}^{(k)}.$$

Возникает вопрос, можно ли использовать лишь две переменные $A$ и $B$ для представления всех $a^{(k)}$ и $b^{(k)}$? Чтобы ответить на этот вопрос, рассмотрим, какие коэффициенты потребуются на последующих шагах обратной подстановки, а именно $x_k = \left( b_i^{(k)} - \sum_{j=k+1}^{n} a_{ij}^{(k)} x_j \right) \Big/ a_{ik}^{(k)}$. Как мы уже видели, при этом можно воспользоваться любым из уравнений с индексом $i = k, \ldots, n$. Если для вычисления $x_k$ выбирается $k$-е уравнение ($i = k$), то доступны как раз те строки $A$ и $B$, которые содержат коэффициенты $a_{ij}^{(k)}$ и $b_i^{(k)}$ для $i = k + 1, \ldots, n$. Таким образом, их можно заменить соответственно на $a_{ij}^{(k+1)}$ и $b_i^{(k+1)}$. Из всего этого следует, что достаточно иметь по одному экземпляру переменных $A$ и $B$, которые будут последовательно хранить коэффициенты $a_{ij}^{(1)}, a_{ij}^{(2)}, \ldots, a_{ij}^{(n)}$ и $b_i^{(1)}, b_i^{(2)}, \ldots, b_i^{(n)}$. Объем памяти, необходимый для хранения этих коэффициентов, резко сокращается. Именно поэтому применение метода исключения Гаусса экономически оправдано на существующих вычислительных машинах. Заметим, что по аналогичным соображениям можно было бы использовать общую память для $x_i$ и $b_i$ и не вводить переменную $X$.

Взглянем на рекуррентные соотношения алгоритма.

Формулы пересчета коэффициентов:

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{kj}^{(k)}}{a_{kk}^{(k)}} a_{ik}^{(k)}, \qquad (4.1)$$

$$b_i^{(k+1)} = b_i^{(k)} - \frac{b_k^{(k)}}{a_{kk}^{(k)}} a_{ik}^{(k)}.$$

Формула получения неизвестных:

$$x_k = \frac{b_i^{(k)} - \sum_{j=k+1}^{n} a_{ij}^{(k)} x_j}{a_{ik}^{(k)}}. \qquad (4.2)$$

При написании программы воспользуемся методом пошаговой детализации. На первом этапе необходимо определиться с основными структурами данных. Учитывая вышесказанное, введем следующие описания переменных:

```
var
  A: array[1..n, 1..n] of real;
  B, X: array[1..n] of real;
```

На самом высоком уровне детализации напишем первый вариант программы в следующем виде:

***Вариант 1:***

```
1.1 begin <<присвоить значения переменным A и B>>;
1.2   for k:=1 to n-1 do begin
1.3     вычислить a^(k+1) и b^(k+1), используя a^(k) и b^(k),
        в соответсвтии с формулами (4.1)
1.4 end;
1.5 k:=n;
1.6 repeat
1.7   <<вычислить x_k в соответствии с (4.2)>>
1.8 until k=0;
1.9 end.
```

Вариант 2 получается в результате детализации инструкции «вычислить $a^{(k+1)}$ и $b^{(k+1)}$, используя $a^{(k)}$ и $b^{(k)}$, в соответствии с (4.1)» (строка 1.3 в варианте 1), которую можно записать следующим образом:

***Вариант 2:***

```
2.1 for i:=k+1 to n do begin
2.2   <<вычислить i-ю строку a^(k+1) и b^(k+1),
      в соответствии с (4.1)>>
2.3 end;
```

В этом месте следует обратить внимание на то, что при вычислении $a_{ij}^{(k+1)}$ и $b_i^{(k+1)}$ по (4.1) множители $a_{kj}^{(k)}/a_{kk}^{(k)}$ и $b_k^{(k)}/a_{kk}^{(k)}$ не зависят от параметра цикла $i$. Следуя основному правилу, по которому нужно избегать повторного вычисления

неменяющихся частей выражений, мы вынесем из инструкции **for** операцию деления на $a_{kk}^{(k)}$. Но возникает вопрос: где хранить результаты деления? Можно ли просто заменить $a_{kj}^{(k)}$ и $b_k^{(k)}$ соответственно на $a_{kj}^{(k)}/a_{kk}^{(k)}$ и $b_k^{(k)}/a_{kk}^{(k)}$? Рассмотрим, как повлияет эта замена на процесс обратной подстановки. Во-первых, умножение всех коэффициентов на один и тот же множитель не изменяет значения неизвестных. Во-вторых, при делении на $a_{kk}^{(k)}$ значение самого $a_{kk}^{(k)}$ становится равным 1, и, следовательно, в процессе обратной подстановки деление становится излишним. Поэтому предлагаемая замена не только допустима, но и желательна. На основании вышесказанного запишем третий вариант программы для фазы исключения неизвестных, который предназначен заменить строки 1.2–1.4 в варианте 1.

***Вариант 3:***

```
3.1 for k:=1 to n-1 do begin
3.2   p:=1/A[k,k];
3.3   for j:=k+1 to n do
3.4     A[k,j]:=p*A[k,j];
3.5   B[k]:=p*B[k];
3.6   for i:=k+1 to n do begin
3.7     <<вычислить a_i^(k+1) и b_i^(k+1), в соответствии с (4.1)>>
3.8   end;
3.9 end;
```

Наконец, при детализации инструкции «вычислить $a_i^{(k+1)}$ и $b_i^{(k+1)}$, в соответствии с (4.1)» (строка 3.7 в варианте 3) следует учитывать, что $A[i, j]$ имеет значение $a_{kj}^{(k)}/a_{kk}^{(k)}$ (на $k$-м шаге).

***Вариант 4:***

```
4.1 for i:=k+1 to n do begin
4.2   for j:=k+1 to n do
4.3     A[i,j]:=A[i,j]-A[i,k]*A[k,j];
4.4   B[i]:=B[i]-A[i,k]*B[k];
4.5 end;
```

Теперь тот же процесс постепенной детализации мы применяем к фазе обратной подстановки. Обратная подстановка реализуется как последовательность вычитаний по (4.2). Заметим только, что деление на $a_{kk}^{(k)}$ уже было выполнено во время фазы исключения неизвестных. Таким образом, строку 1.7 в варианте 1 необходимо заменить следующими строками:

***Вариант 5:***

```
5.1 t:=B[k];
5.2 for j:=k+1 to n do
5.3   t:=t-A[k,j]*X[j];
5.4 X[k]:=t;
```

Обратите внимание, что здесь существенно используется то свойство инструкции **for**, в соответствии с которым не выполняется никаких действий, если конечное значение меньше начального значения параметра цикла.

Ниже приводится окончательный вид программы решения системы линейных уравнений. Заметим, что теперь требуется $n$ шагов исключения неизвестных (вместо $n-1$ шагов), так как на $n$-м шаге выполняется деление $b_n^{(n)}$ на $a_{nn}^{(n)}$.

```
var
  i, j, k: 1..n;
  p, t: real;
  A: array[1..n, 1..n] of real;
  B, X: array[1..n] of real;
begin
  <<присвоить значения переменным A и B>>;
  for k:=1 to n do begin
    p:=1/A[k,k];
    for j:=k+1 to n do
      A[k,j]:=p*A[k,j];
    B[k]:=p*B[k];
    for i:=k+1 to n do begin
      for j:=k+1 to n do
        A[i,j]:=A[i,j]-A[i,k]*A[k,j];
      B[i]:=B[i]-A[i,k]*B[k];
    end;
  end;
  k:=n;
  repeat
    t:=B[k];
    for j:=k+1 to n do
      t:=t-A[k,j]*X[j];
    X[k]:=t
  until k=0;
end.
```

{$X[1]...X[n]$ являются решением системы уравнений}

Особое внимание нужно уделить операции деления, поскольку алгоритм перестает работать, если делитель окажется равным нулю. Это тем более важно, если арифметические операции выполняются с конечной точностью, так как даже близкий к нулю делитель может привести к ошибке или в лучшем случае к совершенно обескураживающим результатам. То, что перестановка строк (уравнений) и столбцов $A$ и $B$ не влияет на значения неизвестных, позволяет нам в такой последовательности производить исключение неизвестных, чтобы вообще избежать деления на нуль или близкое к нулю значение. Делители называются осевыми элементами. В качестве осевого элемента мы можем выбрать компоненту $a_{ij}^{(k)}$ с наибольшим абсолютным значением. Разумеется, алгоритм при этом усложняется, но без поиска осевого элемента невозможно, как правило, обойтись, если мы хотим иметь удовлетворительную точность и надежность вычислений. Если на некотором шаге исключения не удается найти отличный от нуля осевой элемент, то такая система уравнений называется сингулярной, т. е. она имеет не единственное решение. Если же нет делителя, который существенно отличается от нуля, то такая система является *плохо обусловленной*.

Краткий анализ алгоритма показывает, что в самом внутреннем цикле операции

$$A[i, j] = A[i, j] - A[i, k] \times A[k, j]$$

выполняются

$$(n-1)^2 + (n-2)^2 + \ldots + 2^2 + 1^2 = \frac{1}{6}(2n^3 - 3n^2 + n)$$

раз. То есть при решении системы линейных алгебраических уравнений методом Гаусса объем вычислений растет как третья степень числа порядка системы $n$.

## 4.2 Метод прогонки

Данный метод применяется в случае трехдиагональной матрицы $A$. В этом случае учитывается особенность структуры матрицы, а систему можно записать в виде:

$$a_i y_{i-1} + c_i y_i + b_i y_{i+1} = f_i,\ a_0 = b_n = 0,\ i = 0 \ldots n. \tag{4.3}$$

В развернутом виде это выглядит следующим образом:

$$\begin{pmatrix} c_0 & b_0 & 0 & 0 & \ldots & 0 & 0 \\ a_1 & c_1 & b_1 & 0 & \ldots & 0 & 0 \\ 0 & a_2 & c_2 & b_2 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & a_{n-1} & c_{n-1} & b_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & a_n & c_n \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \ldots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \ldots \\ f_{n-1} \\ f_n \end{pmatrix} \tag{4.4}$$

Путем несложных манипуляций, в частности применив метод исключения Гаусса, систему можно преобразовать к виду:

$$\begin{pmatrix} 1 & -\alpha_0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & -\alpha_1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & -\alpha_2 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 0 & 1 & -\alpha_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \ldots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \ldots \\ \beta_{n-1} \\ \beta_n \end{pmatrix} \tag{4.5}$$

Однако данный процесс требует хранения всей матрицы $A$, что в случае матрицы большой размерности не всегда возможно. А учитывая, что матрица имеет трехдиагональный вид, то метод Гаусса большую часть времени будет выполнять операции с нулевыми элементами.

Но, допустим, матрица вида (4.5) получена, что нам это дает? Из нее получаем рекуррентные соотношения:

$$y_i = \alpha_i y_{i+1} + \beta_i,\ i = 0, 1, \ldots, n-1 \tag{4.6}$$

или

$$y_{i-1} = \alpha_{i-1} y_i + \beta_{i-1}, \tag{4.7}$$

Пока нам известны только $\alpha_1$ и $\beta_1$ ($\alpha_1 \equiv -b_0/c_0$; $\beta_1 \equiv f_0/c_0$), а $\alpha_2, \ldots, \alpha_n$ и $\beta_2, \ldots, \beta_n$ — пока неизвестны.

Подставив уравнение (4.7) в уравнение (4.3), получаем:

$$\alpha_i(\alpha_{i-1}y_i + \beta_{i-1}) + c_i y_i + b_i y_{i+1} = f_i,\ i = 1, \ldots, n-1$$

Приведя подобные получаем:

$$y_i = -\frac{b_i}{a_i\alpha_{i-1} + c_i}y_{i+1} + \frac{f_i - a_i\beta_{i-1}}{a_i\alpha_{i-1} + c_i}.$$

Сравнив полученное уравнение с (4.6), обнаружим, что:

$$\alpha_i = \frac{-b_i}{\alpha_{i-1}a_i + c_i};\quad \beta_i = \frac{f_i - \alpha_i\beta_{i-1}}{\alpha_{i-1}a_i + c_i}. \tag{4.8}$$

Таким образом, видим, что нет необходимости в преобразовании (4.4) к виду (4.5), так как все коэффициенты (4.8) можно получить, имея только $\alpha_1$, $\beta_1$ и 3 одномерных массива размерности $n$ каждый (или один двумерный, размерности $3\times n$). Соотношения, определяющие коэффициенты (4.8), называются формулами прямой прогонки.

После того как по рекуррентным соотношениям (4.8) найдем все значения прогоночных коэффициентов: $(\alpha_2, \beta_2)$, $(\alpha_3, \beta_3)$, …, $(\alpha_{n-1}, \beta_{n-1})$, переходим ко второму этапу, называемому — обратная прогонка.

Этот этап начинается с вычисления $y_n$. Его значение находим, разрешив систему уравнений, составленную из уравнения с последними полученными прогоночными коэффициентами и последнего уравнения исходной системы:

$$\begin{cases} y_{n-1} = \alpha_{n-1}y_n + \beta_{n-1}; \\ f_n = a_n y_{n-1} + y_n c_n. \end{cases}$$

Разрешив систему относительно $y_n$, получим:

$$y_n = \frac{-a_n\beta_{n-1} + f_n}{c_n + a_n\alpha_{n-1}}.$$

Здесь все коэффициенты известны, следовательно, значение определено. Все остальные значения $y_i$ ($i = n-1, n-2, \ldots, 0$) находим по рекуррентной формуле (4.6).

## 4.3 Итерационные методы решения СЛАУ

При изучении итерационных методов используются понятия нормы вектора и нормы матрицы. Рассмотрим основные виды норм. Если в пространстве векторов $\mathbf{x} = (x_1, \ldots, x_n)^T$ введена норма $\|\mathbf{x}\|$, то согласованной с ней нормой в пространстве матриц A называют норму

$$\|\mathbf{A}\| = \sup_{\mathbf{x}\neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}.$$

Наиболее используемы в пространстве векторов следующие нормы:

$$\|\mathbf{x}\|_\infty = \max_j |x_j|$$
$$\|\mathbf{x}\|_1 = \sum_j |x_j|$$
$$\|\mathbf{x}\|_2 = \sqrt{\sum_j |x|^2} = \sqrt{(x, x)},$$

а согласованными с ними, нормами в пространстве матриц являются соответственно нормы:

$$\|\mathbf{A}\|_\infty = \max_i \left( \sum_j |a_{ij}| \right),$$
$$\|\mathbf{A}\|_1 = \max_j \left( \sum_j |a_{ij}| \right),$$
$$\|\mathbf{A}\|_2 = \sqrt{\max_i \lambda^i_{A^T A}},$$

где $\lambda^i_{A^T A}$ — собственные значения матрицы $\mathbf{A}^T\mathbf{A}$.

### 4.3.1 Метод простых итераций

Суть метода состоит в последовательном выполнении следующих процедур.

1) Исходная задача $A \times \mathbf{x} = \mathbf{b}$ преобразуется к равносильному виду: $\mathbf{x} = c \times \mathbf{x} + \mathbf{d}$, где $c = \{c_{ij}\}$ — квадратная матрица, $\mathbf{d} = d\{d_i\}$ — вектор, $i, j = 1, \ldots, n$. Такое преобразование можно осуществить различными способами, однако при осуществлении преобразования следует помнить о том, что для обеспечения сходимости норма матрицы должна быть меньше единицы $\|c\| < 1$.

   Например, систему:

$$1.02 \times x_1 - 0.15 \times x_2 = 2.7$$
$$0.8 \times x_1 + 1.05 \times x_2 = 4$$

   можно записать в форме:

$$x_1 = -0.02 \times x_1 + 0.15 \times x_2 + 2.7$$
$$x_2 = -0.8 \times x_1 - 0.05 \times x_2 + 4$$

   для которой $\|c\|_\infty = \max\{0.17, 0.85\} = 0.85 < 1$.

2) Выбирается начальное приближение решения, обычно в качестве начального значения вектора решения берут вектор $\mathbf{d}$, $\mathbf{x}^{(0)} = \mathbf{d}$. И затем многократно выполняют действия по уточнению решения в соответствии с рекуррентной формулой:

$$\mathbf{x}^{(k+1)} = c \times \mathbf{x}^{(k)} + \mathbf{d}, \; k = 0, 1, \ldots.$$

3) Итерации по уточнению решения заканчиваются при выполнении условия:

$$\frac{\|c\|}{1-\|c\|}\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\|<\varepsilon,$$

где $\varepsilon > 0$ — заданная точность, которую необходимо достигнуть при решении задачи.

Ниже приведен примерный шаблон программы, реализующий метод простых итераций, в котором расписана основная часть алгоритма. Для конкретной задачи данный шаблон естественно требует уточнения.

***Метод простых итераций вычисления решения СЛАУ***

- *n — количество неизвестных и уравнений системы*
- *c[n][n] — матрица коэффициентов после преобразования*
- *d[n] — вектор свободных членов после преобразования*
- *xk[n] — вектор решения на шаге k*
- *xk1[n] — вектор решения на шаге k + 1*
- *e — точность*

```
type
  matrix=array[1..n, 1..n] of real;
  vect=array[1..n] of real;
function norm_matrix(var m: matrix): real;
{
  Для каждого типа нормы определяется по своему
}
function norm_vect(var y: vect): real;
{
  Для каждого типа нормы определяется по своему
}
var
  ...
  i, j: integer;
  tmp, norm: real;
  t: vect;
begin
  ...
  for i:=1 to n do
    xk1[i]:=d[i];
  repeat
    for i:=1 to n do
      xk[i]:=xk1[i];
    for i:=1to n do
    begin
      xk1[i]:=0;
      for j:=1 to n do
        xk1[i]:=xk1[i]+c[i][j]*xk[j];
```

```
        xk1[i]:=xk1[i]+d[i];
        t[i]:=xk1[i]-xk[i];
      end;
      tmp:=norm_matrix(c);
      norm:=norm_vect(t)*tmp/(1-tmp);
    until norm<e;
end.
```

### 4.3.2 Метод итераций

Для решения частичной проблемы собственных чисел для матриц практически любой размерности удобнее всего использовать метод итераций. Следует заметить, что данный метод позволяет получить собственное значение приближенно.

**Пример**

Для демонстрации метода найдем собственное значение и соответствующий ему собственный вектор для матрицы $\mathbf{A} = \begin{pmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 1 & 3 \end{pmatrix}$, с точностью $\varepsilon = 0.1$.

1) Положим начальное приближение $\mathbf{X}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

2) Найдем следующее приближение:

$$\mathbf{X}_1 = \mathbf{A} \times \mathbf{X}_0 = \begin{pmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 1 & 3 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ 6 \end{pmatrix}, \quad \lambda_1 = \frac{8}{1} = 8, \quad |\lambda_1 - \lambda_0| = 8 > 0.1.$$

3) Найдем следующее приближение:

$$\mathbf{X}_2 = \mathbf{A} \times \mathbf{X}_1 = \begin{pmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 1 & 3 \end{pmatrix} \times \begin{pmatrix} 8 \\ 6 \\ 6 \end{pmatrix} = \begin{pmatrix} 58 \\ 38 \\ 40 \end{pmatrix}, \quad \lambda_2 = \frac{58}{8} = 7.25, \quad |\lambda_2 - \lambda_1| = 0.75 > 0.1.$$

4) Продолжаем уточнение:

$$\mathbf{X}_3 = \mathbf{A} \times \mathbf{X}_2 = \begin{pmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 1 & 3 \end{pmatrix} \times \begin{pmatrix} 58 \\ 38 \\ 40 \end{pmatrix} = \begin{pmatrix} 408 \\ 250 \\ 274 \end{pmatrix}, \quad \lambda_3 = \frac{408}{58} = 7.034, \quad |\lambda_3 - \lambda_2| = 0.116 > 0.1.$$

5) Продолжаем уточнение:

$$\mathbf{X}_4 = \mathbf{A} \times \mathbf{X}_3 = \begin{pmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 1 & 3 \end{pmatrix} \times \begin{pmatrix} 408 \\ 250 \\ 274 \end{pmatrix} = \begin{pmatrix} 2838 \\ 1682 \\ 1888 \end{pmatrix},$$

$$\lambda_4 = \frac{2838}{408} = 6.9559, \quad |\lambda_4 - \lambda_3| = 0.078 > 0.1.$$

Заданная точность достигнута, поэтому можем взять в качестве собственного числа $\lambda_4 = 6.9559$, а в качестве собственного вектора $\mathbf{X}_4 = \begin{pmatrix} 2838 \\ 1682 \\ 1888 \end{pmatrix}$. Для удобства с полученным вектором лучше произвести нормировку. В нашем случае $\mathbf{X} = \frac{1}{2838} \times \begin{pmatrix} 2838 \\ 1682 \\ 1888 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.5927 \\ 0.6652 \end{pmatrix}$.

. . . . . .

Шаблон программы, реализующей данный метод, представлен ниже.

***Метод итераций, решения частичной проблемы собственных чисел***

- *n — размерность матрицы*
- *c*[*n*][*n*] *— матрица*
- *xk*[*n*] *— собственный вектор на шаге k*
- *xk*1[*n*] *— собственный вектор на шаге k* + 1
- *e — точность*
- *lk — собственное значение на k-ом шаге*
- *lk*1 *— собственное значение на k* + 1*-ом шаге*

```
type
  matrix=array[1..n, 1..n] of real;
  vect=array[1..n] of real;
const
  n=10;
procedure mult(var a: matrix; var x1, x0: vect);
var
  i, j: integer;
begin
  for i:=1 to n do
  begin
    x1[i]:=0;
    for j:=1 to n do
      x1[i]:=x1[i]+a[i][j]*x0[j];
  end;
end;
var
  ...
  i, j: integer;
  t, lk, lk1: real;
begin
  ...
  for i:=1 to n do
    xk[i]:=1;
```

```
    mult(c,xk1,xk);
    lk1:=xk1[1]/xk[1];
    repeat
      lk:=lk1;
      for i:=1 to n do
        xk[i]:=xk1[i];
      mult(c,xk1,xk);
      lk1:=xk1[i]/xk[i];
    until abs(lk-lk1)<e;
    t:=xk1[1];
    for i:=1 to n do
      xk1[i]:=xk1[i]/t;
  end.
```

# Глава 5

# ВЫЧИСЛЕНИЕ ПОЛИНОМОВ

Некоторые программы многократно вычисляют определенные полиномы для различных значений их аргументов. Поэтому важно уметь быстро вычислять полиномы.

Вычисление полинома:

$$P(x) = a_1 x^n + a_2 x^{n-1} + \ldots + a_n x + a_{n+1}$$

посредством следующего фрагмента Паскаль-программы ($power(x, n)$ — функция $x^n$)

```
p:=a[n+1];
for i:=1 to n do
  p:=p+a[i]*power(x,n-i+1);
```

требует $n(n+1)/2$ умножений и $n$ сложений. Простой метод, называемый схемой Горнера, состоит в перезаписи $P(x)$ в виде:

$$P(x) = a_{n+1} + x\Big(a_n + x\big(a_{n-1} + x\,(\ldots(a_2 + xa_1)\ldots)\big)\Big).$$

Это легко программируется. Например,

```
p:=a[1];
for i:=1 to n do
  p:=p*x+a[i+1]
```

Схема Горнера требует только $n$ умножений и $n$ сложений с плавающей точкой. Известно, что схема Горнера является оптимальной среди методов, переупорядочивающих полином для быстрого вычисления и при этом не делающих значительных вычислений в процессе переупорядочения. Таким образом, если заданы коэффициенты полинома и аргумент $x$, то, вообще говоря, нельзя вычислить полином за меньшее число сложений и умножений, чем для схемы Горнера.

# Глава 6

# ИНТЕРПОЛЯЦИЯ

Получаемые при компьютерных вычислениях, в экспериментальных исследованиях или задаваемые при проектировании сеточные (табличные) функции:

$$y_i = f(x_i),\ x_i \in [a, b],\ i = 0, \ldots, n,$$

малоинформативны. Они определены только в узлах $x_i$ сетки, а их значения в промежуточных точках, а также значения производных в узлах не известны, от них нельзя вычислить интегралы классическими способами.

Однако значения функции должны быть известны при любом значении $x \neq x_i$, а в самих узлах $x_i$ требуется знать так же первые и вторые производные, поэтому сеточные функции $y_i = f(x_i)$ необходимо восполнять. Данная проблема решается с помощью методов теории приближений.

Целью задачи интерполяции является вычисление значения функции в произвольной точке $x_* \in [x_0, x_n]$, если же $x_* \notin [x_0, x_n]$, то такая задача называется экстраполяцией.

Так как изначально вид функции неизвестен, применяют различные подходы к выбору интерполяционной функции. Как правило, ее задают в виде алгебраического многочлена. Широко применяются многочлены Лагранжа, Ньютона и их модификации. Но в вычислительной практике при глобальном способе аппроксимации высокие степени интерполяционных многочленов использовать нецелесообразно.

Для проведения гладких кривых через узловые значения функции чертежники используют упругую металлическую линейку, совмещая ее с узловыми точками. Математическая теория подобной аппроксимации называется теорией сплайн-функций.

Рассмотрим наиболее распространённый вариант интерполяции кубическими сплайнами.

На рис. 6.1 представлена интерполяция кубическим сплайном функции заданной таблично:

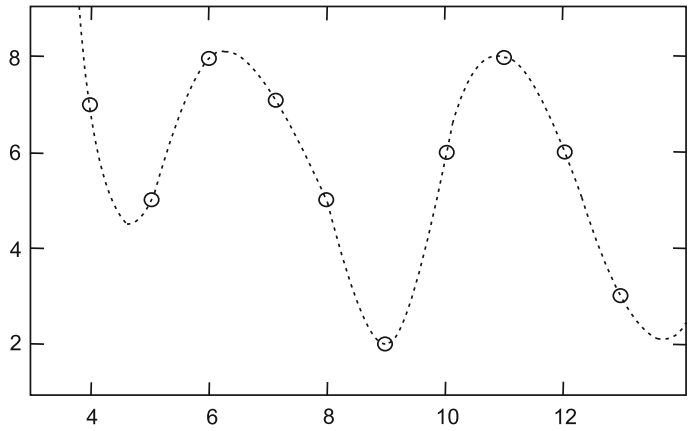



Рис. 6.1 – Кубическая интерполяция

Таблица 6.1

| $x_i$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| $y_i$ | 7 | 5 | 8 | 7 | 5 | 2 | 6 | 8 | 6 | 3 |

Используя законы упругости, можно установить, что недеформируемая линейка между соседними узлами проходит по линии, удовлетворяющей уравнению:

$$S^{(4)}(x) = 0. \tag{6.1}$$

Функцию S(x) будем использовать для аппроксимации зависимости $f(x)$, заданной в узлах $x_0, x_1, \ldots, x_n$ значениями $f_0, f_1, \ldots, f_n$.

Если в качестве $S(x)$ выбираем полином, то в соответствии с уравнением (6.1) степень полинома должна быть не больше третьей. Этот полином называется кубическим сплайном, который на интервале $x \in [x_{i-1}, x_i]$ записывается в виде:

$$S_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3;\ x_{i-1} \leqslant x \leqslant x_i,\ i = 1\ldots n.$$

где $a_i$, $b_i$, $c_i$, $d_i$ — коэффициенты, определяемые из дополнительных условий (для упрощения обозначим $h_i = x - x_i$):

$$\begin{aligned}
&S_i(x_{i-1}) = a_i = f(x_{i-1}), \quad n \text{ уравнений}\\
&S_i(x_i) = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f(x_i), \quad n \text{ уравнений}\\
&S_i'(x_i) = S_{i+1}'(x_i), \quad n-1 \text{ уравнений}\\
&S_i''(x_i) = S_{i+1}''(x_i). \quad n-1 \text{ уравнений}
\end{aligned}$$

Помним, что в результате дифференцирования производные будут выглядеть следующим образом:

$$\begin{aligned}
&S_i'(x) = b_i + 2c_i(x - x_{i-1}) + 3d_i(x - x_{i-1})^2.\\
&S_i''(x) = 2c_i + 6d_i(x - x_{i-1}).
\end{aligned}$$

Для полной определенности системы не хватает двух уравнений, их получают из условий на границах отрезка. Допустим, нам известны первые производные $f'(a) = f_a; f'(b) = f_b$.

В итоге получаем систему из $4 \times n$ уравнений для определения коэффициентов:

$$a_i = f(x_{i-1});\ i = 1, \ldots, n$$
$$a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = f(x_i);\ i = 1, \ldots, n$$
$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1};\ i = 1, \ldots, n-1$$
$$c_i + 3d_i h_i = c_{i+1};\ i = 1, \ldots, n-1$$
$$b_1 = f_a.$$
$$b_n + 2c_n h_n + 3d_n h_n^2 = f(b).$$

Данная система является полной, и ее можно решить имеющимися средствами. Однако гораздо выгоднее систему несколько преобразовать, выразив коэффициенты $a_i$, $b_i$, и $d_i$ через $c_i$. Вместе с граничными условиями получим систему из $n$ уравнений для определения коэффициентов $c_i$. Так же примечателен тот факт, что в полученной в результате таких преобразований системе в каждое уравнение входит только три неизвестных с последовательными индексами $c_{i-1}$, $c_i$, $c_{i+1}$. Следовательно, матрица этой системы является трехдиагональной. И для решения такой системы эффективно использовать метод прогонки.

# Глава 7

# ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

К численному интегрированию прибегают в том случае, когда невозможно через элементарные функции аналитически записать первообразную интеграла либо такая запись существенно затруднена.

В большинстве методы вычисления определенных интегралов состоят в замене подынтегральной функции аппроксимирующей, для которой гораздо легче записать первообразную в элементарных функциях.

Используемые в практических вычислениях методы можно разделить на группы по способу аппроксимации подынтегральной функции.

При вычислении интеграла следует помнить, что для получения формул численного интегрирования на некотором отрезке $[a, b]$ достаточно построить квадратурную формулу для интеграла на элементарном отрезке $[x_i, x_{i+1}]$, а затем ее просуммировать, т. к.

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx.$$

## 7.1 Метод прямоугольников

В данной группе методов подынтегральную функцию заменяют полиномом нулевой степени, то есть константой. Подобная замена неоднозначна, так как константу можно выбрать равную любому значению подынтегральной функции в области интегрирования. Таким образом, интеграл аппроксимируется прямоугольником, за что метод и получил свое название, со сторонами, равными значению подынтегральной функции и длине интервала интегрирования.

Для уменьшения ошибки, при вычислении определенного интеграла интервал интегрирования стараются разбить на такое количество элементарных интервалов, которое даст нам необходимую точность. В силу конечности вычислительных возможностей компьютера, мы не можем устремить длину элементарного интервала

интегрирования к нулю. Но на практике это и не требуется. Обычно поступают следующим образом:

- Разбивают интервал интегрирования на небольшое количество элементарных интервалов.
- Вычисляют значение определенного интеграла для этого количества элементарных интервалов.
- Увеличивают количество элементарных интервалов интегрирования, тем самым уменьшая их длину, и уже с этим количеством интервалов опять вычисляют значение интеграла.

Процедуру увеличения числа элементарных интервалов интегрирования и вычисления значения определенного интеграла продолжают до тех пор, пока абсолютное значение разности интегралов, полученных на соседних итерациях, не окажется меньше заданной точности. Однако слишком малое значение точности может потребовать разбиение интервала интегрирования на слишком большое число элементарных интегралов, а это, в свою очередь, повлечет накопление ошибки округления.

В зависимости от способа выбора константы на интервале интегрирования различают методы левых, правых и центральных прямоугольников. В этих методах в качестве аппроксимирующей константы выбирается значение подынтегральной функции соответственно на левой, правой границе или в центре интервала интегрирования.

Ниже представлен вариант процедуры вычисления определенного интеграла на отрезке $[a, b]$ с количеством элементарных интервалов $n$ методом центральных прямоугольников.

***Метод центральных прямоугольников для вычисления определенного интеграла.***

- *a, b — границы интервала интегрирования,*
- *n — число разбиений,*
- *s — результат (значение определенного интеграла),*
- *f — подынтегральная функция (должна быть определена выше).*

```
procedure rect(a, b: real; n: integer; var s: real);
var
  h, x: real; {h — шаг интегрирования,
    x — текущее значение переменной интегрирования}
  i: integer;
begin
  s:=0;
  h:=(b-a)/n; {определение шага интегрирования}
  x:=a+h/2;
  for i:=1 to n do begin
    s:=s+f(x);
    x:=x+h;
  end;
  s:=s*h;
end;
```

## 7.2 Метод трапеций

Для реализации этого метода подынтегральную функцию заменяют полиномом первой степени. Таким образом, интеграл на интервале интегрирования $[x_i, x_i + h]$ приближенно представляется площадью трапеции, опять же это и дало название методу, ограниченной прямыми: $x = x_i$, $x = x_i + h$, $y = 0$ и прямой, проходящей через точки $\big(x_i, f(x_i)\big)$ и $\big(x_i + h, f(x_i + h)\big)$.

Просуммировав по всему отрезку, получим *составную формулу трапеций*:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{i=1}^{n}(y_{i-1} + y_i).$$

Этот метод дает несколько неожиданный на первый взгляд результат. Несмотря на то, что аппроксимация подынтегральной функции в данном методе осуществляется полиномом первой степени, в отличие от метода прямоугольников, где полином имеет нулевую степень, точность данного метода имеет тот же порядок, что и метод центральных прямоугольников. Но более удивительным покажется тот факт, что метод трапеций имеет в два раза большую по абсолютной величине погрешность по сравнению с погрешностью метода центральных прямоугольников.

Ниже представлен вариант процедуры вычисления определенного интеграла на отрезке $[a, b]$ с количеством элементарных интервалов интегрирования $n$ методом трапеций.

***Метод трапеций для вычисления определенного интеграла.***

- *a, b — границы интервала интегрирования,*
- *n — число разбиений,*
- *s — результат (значение определенного интеграла),*
- *f — подынтегральная функция (должна быть определена выше).*

```
procedure trap(a, b: real; n: integer; var s: real);
var
  h, x: real; {h — шаг интегрирования,
    x — текущее значение переменной интегрирования}
  i: integer;
begin
  s:=(f(a)+f(b))/2;
  h:=(b-a)/n; {определение шага интегрирования}
  x:=a+h;
  for i:=1 to n-1 do begin
    s:=s+f(x);
    x:=x+h;
  end;
  s:=s*h;
end;
```

## 7.3 Метод Симпсона

В методе Симпсона подынтегральную функцию аппроксимируют полиномом второй степени, то есть параболой, проходящей через три точки элементарного интервала интегрирования: $x_{i-1}$, $x_{i-1/2}$, $x_i$.

Значение определенного интеграла на элементарном интервале интегрирования приближенно заменяется площадью криволинейной трапеции.

Составная формула Симпсона определяется как:

$$\int_a^b f(x)\,dx \approx \frac{h}{6}\sum_{i=1}^{n}(y_{i-1} + 4y_{i-\frac{1}{2}} + y_i) = \frac{h}{6}\left[y_0 + y_n + 2\sum_{i=1}^{n-1} y_i + 4\sum_{i=1}^{n} y_{i-\frac{1}{2}}\right].$$

Метод Симпсона имеет четвертый порядок точности. Метод Симпсона позволяет получить высокую точность, если четвертая производная подынтегральной функции не слишком велика. В противном случае методы второго порядка могут дать большую точность, чем метод Симпсона.

Ниже представлен вариант процедуры вычисления определенного интеграла на отрезке $[a, b]$ с количеством элементарных интервалов интегрирования $n$ методом Симпсона.

***Метод Симпсона для вычисления определенного интеграла.***

- *a, b — границы интервала интегрирования,*
- *n — число разбиений,*
- *s — результат (значение определенного интеграла),*
- *f — подынтегральная функция (должна быть определена выше).*

```
procedure simpson(a, b: real; n: integer; var s: real);
var
  h, h2, x: real; {h — шаг интегрирования,
    h2 — полушаг интегрирования,
    x — текущее значение переменной интегрирования}
  i: integer;
begin
  h:=(b-a)/n; {определение шага интегрирования}
  h2:=h/2;
  x:=a;
  s:=(f(a)+f(b))/2+2*f(a+h2);
  for i:=1 to n-1 do begin
    x:=x+h;
    s:=s+f(x)+f(x+h2);
  end;
  s:=s*h/3;
end;
```

# Глава 8

# ЧИСЛЕННОЕ РЕШЕНИЕ ЗАДАЧИ КОШИ ДЛЯ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

Обыкновенные дифференциальные уравнения широко применяются для математического моделирования процессов и явлений в различных областях науки и техники. Переходные процессы в радиотехнических цепях, движение космических объектов, модели экономического развития исследуются с помощью обыкновенных дифференциальных уравнений.

Следует помнить, что лишь немногие дифференциальные уравнения решаются аналитическим способом, поэтому очень важно иметь инструмент для решения подобного рода задач. Более значимым является инструмент решения систем обыкновенных дифференциальных уравнений, так как реальные процессы редко описываются уравнением первого порядка, а так же редко зависимости бывают только от одной переменной. Дифференциальное уравнение порядка $n$ можно свести к системе из $n$ уравнений первого порядка, поэтому большинство методов ориентировано на решение системы дифференциальных уравнений.

## 8.1 Метод Эйлера

Для применения метода Эйлера система дифференциальных уравнений должна быть представлена в каноническом виде:

$$\frac{dy_k(x)}{dx} = f_k(x, y_1, y_2, \ldots, y_n),$$

где $k = 1, 2, \ldots, n$.

В данном методе на каждом шаге строится приближение к решению, причем значение, полученное на предыдущем шаге, является начальным значением для нахождения следующего приближения.

***Метод Эйлера для системы** n **обыкновенных дифференциальных уравнений первого порядка, записанной в векторном виде:***

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}).$$

- *ntax — максимально возможное число уравнений,*
- *x — независимая переменная,*
- **y** *— вектор зависимых переменных (искомое решение),*
- **f** *— вектор правых частей,*
- *a — начальное значение независимой переменной,*
- *b — конечное значение независимой переменной,*
- *h — шаг интегрирования.*

```
const
  nmax=8;
type
  vec=array[1..nmax] of real;
var
  h, a, b: real;
procedure der(x: real; y: vec; var f: vec);
begin
{
  Процедурные вычисления правых частей.
}
end;

procedure euler(n: integer; x, h: real; var y: vec);
var
  i: integer;
  f: vec;
begin
  der(x,y,f);
  for i:=1 to n do
    y[i]:=y[i]+h*f[i];
end;
```

## 8.2 Метод Рунге — Кутта

Методы Рунге — Кутта рассмотрим на примере метода четвертого порядка:

$$y_{k+1} = y_k + \frac{1}{6}(m_1 + 2m_2 + 2m_3 + m_4),$$

где

$$m_1 = hf(t_k, y_k); \quad m_2 = hf\left(t_k + \frac{h}{2}, y_k + \frac{m_1}{2}\right);$$
$$m_3 = hf\left(t_k + \frac{h}{2}, y_k + \frac{m_2}{2}\right); \quad m_4 = hf(t_{k+1}, y_k + m_3).$$

***Метод Рунге — Кутта четвертого порядка точности для системы*** *n* ***обыкновенных дифференциальных уравнений первого порядка, записанной в векторном виде:***

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}).$$

- *nmax — максимально возможное число уравнений,*
- *x — независимая переменная,*
- **y** *— вектор зависимых переменных (искомое решение),*
- **f** *— вектор правых частей,*
- *a — начальное значение независимой переменной,*
- *b — конечное значение независимой переменной,*
- *h — шаг интегрирования.*

```
const
  nmax=8;
type
  vec=array[1..nmax] of real;
var
  h, a, b: real;
procedure der(x: real; y: vec; var f: vec);
begin
{
  Процедурные вычисления правых частей.
}
end;

procedure rk4(n: integer; x, h: real; var y: vec);
var
  i, j: integer;
  h1, h2, q: real;
  y0, y1, f: vec;
begin
  h1:=0; h2:=h1/2;
  for i:=1 to n do begin
    y0[i]:=y[i]; y1[i]:=y[i];
  end;
  for j:=1 to 4 do begin
    der(x+h1,y,f);
    if j=3 then h1:=h else h1:=h2;
    for i:=1 to n do begin
```

```
        q:=h1*f[i]; y[i]:=y0[i]+q;
        if j=2 then q=2*q;
        y1[i]=y1[i]+q/3;
      end;
    end;
    for i:=1 to n do
      y[i]:=y1[i];
end;
```

# Глава 9

# РЕШЕНИЕ ТРАНСЦЕНДЕНТНЫХ УРАВНЕНИЙ

Методы решения трансцендентных уравнений включают два этапа. Сначала определяют интервал неопределенности, на котором имеется только один корень уравнения. Затем происходит процедура уточнения корня. Рассматриваемые ниже методы осуществляют процедуру уточнения корня на интервале неопределенности.

## 9.1 Метод половинного деления

Геометрическое представление решения уравнения $f(x) = 0$ это точка пересечения графика функции $f(x)$ с осью $Ox$. Так как интервал неопределенности содержит только один корень уравнения, то на границах этого интервала значения функции $f(x)$ имеют разные знаки. Делим этот интервал пополам и выбрасываем из рассмотрения ту половину интервала, для которой значение функции на границе интервала совпадает по знаку со значением функции в средней точке интервала. Так продолжаем до тех пор, пока длина интервала неопределенности не станет меньше заданной наперед величины либо значение функции в средней точке интервала неопределенности не окажется в переделах заданной точности.

***Метод половинного деления (метод бисекции) для решения трансцендентного уравнения*** $f(x) = 0$**.**

- $a$ и $b$ — *нижняя и верхняя границы промежутка, на котором ищется решение (знаки величин* $f(a)$ *и* $f(b)$ *должны быть различны),*
- $e$ и $e1$ — *оценка точности получаемого решения, итерационный процесс прекращается при сближении a и b меньше, чем e, или приближении* $f(x)$ *к нулю меньше, чем на* $e1$*,*
- $x$ — *искомый корень.*

```
function f(x: real): real;
begin
  {конкретизация функции f(x)}
end;
procedure bisec(a, b, e, e1: real; var x: real);
  var
    r: real;
  function sign(x: real): integer;
  begin
    sign:=0;
    if x<0 then sign:=-1;
    if x>0 then sign:=1;
  end;
begin
  while b-a>e do begin
    x:=(a+b)/2;
    r:=f(x);
    if abs(r)<e1 then exit;
    if sign(f(a))=sign(f(b)) then
      a:=x;
    else
      b:=x;
  end;
end;
```

## 9.2 Метод Ньютона

Для метода Ньютона необходимо знать приблизительное значение корня уравнения $f(x) = 0$. В качестве следующего приближения к решению берется точка пересечения касательной, проведенной к графику функции в точке начального приближения и оси *Ox*. Процесс уточнения заканчивается, когда значение функции $f(x)$ в точке очередного приближения окажется меньше заданной точности. Существенным недостатком данного метода является тот факт, что нам необходимо знать первую производную функции $f(x)$.

***Метод Ньютона для решения трансцендентного уравнения*** $f(x) = 0$**.**

- *a — начальное приближение,*
- *e — оценка точности получаемого решения, итерационный процесс прекращается при приближении* $f(x)$ *к нулю меньше, чем на e,*
- *x — искомый корень,*
- $g(x) = f(x)/f'(x)$.

```
function g(x: real): real;
begin
  {конкретизация выражения f(x)/f'(x)}
end;
```

```
procedure newton(var x: real; e: real);
var
  g1: real;
begin
  repeat
    g1:=g(x);
    x:=x-g1;
  until abs(g1)<e;
end;
```

## 9.3 Метод секущих

Метод секущих аналогичен методу Ньютона за исключением того, что не требуется знать вид первой производной функции $f(x) = 0$. Для реализации метода необходимо знать два начальных приближения корня уравнения $x_1$ и $x_2$. Через точки $(x_1, f(x_1))$ и $(x_2, f(x_2))$ проводится секущая к графику функции и в качестве следующего приближения берется точка пересечения полученной секущей и оси *Ox*.

***Метод секущих для решения трансцендентного уравнения*** $f(x) = 0$**.**

- *x, x0 — два начальных приближения,*
- *e — оценка точности получаемого решения, итерационный процесс прекращается при приближении $f(x)$ к нулю меньше, чем на e,*
- *x — искомый корень.*

```
function f(x: real): real;
begin
  {конкретизация функции f(x)}
end;
procedure secant(var x0, x: real; e: real);
var
  d, y, r: real;
begin
  r:=x-x0;
  d:=f(x0);
  repeat
    y:=f(x);
    r:=r/(d-y)*y;
    d:=y;
    x:=x+r;
  until abs(r)<e;
end;
```

# Глава 10

# УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

В рамках курса необходимо выполнить две лабораторные работы. Каждая лабораторная работа состоит из двух частей. В каждой лабораторной работе необходимо написать программы, реализующие соответствующие заданию методы. По каждой лабораторной работе необходимо оформить отчет, в котором необходимо описать реализуемый метод, привести результаты работы программы и текст программы. Так же на проверку необходимо представить файлы, содержащие исходные тексты программ.

Лабораторная работа №1 включает в себя задание на решение систем линейных алгебраических уравнений (Задание №1) и задание на численное интегрирование (Задание №2).

Лабораторная работа №2 включает в себя задание на численное решение задачи Коши (Задание №3) и задание на решение трансцендентного уравнения (Задание №4).

Варианты заданий рассчитываются по общим правилам.

Для выполнения лабораторных работ рекомендуется использовать язык программирования Free Pascal.

# Глава 11

# ВАРИАНТЫ ЗАДАНИЙ

## Задание №1

### Вариант 1

$$\begin{cases} -2u - 2v + 2x + 11y - 6z = -58 \\ -4u + 6v + 9x + 5y - 8z = 33 \\ 5u + 5v - 5x + 9y + 2z = -108 \\ 10u - 2v - 4x + 2y - 5z = -79 \\ 9u + 4v - 2x + 7y - 2z = -107 \end{cases}$$

### Вариант 2

$$\begin{cases} -8u - 3v + 7x + 10y - 3z = 202 \\ 5u + 6v + 10x + 6y + 2z = 80 \\ 11u - 2v + 8x + 5y + 8z = -36 \\ -5u - 8v + 11x + 5y - 6z = 145 \\ 4u - 7v + 9x + 2y + 4z = -4 \end{cases}$$

## Вариант 3

$$\begin{cases} 11u - 6v + 11x - 6y - 2z = 82 \\ -2u - 5v + 9x + 2y + 3z = 23 \\ -3u + 5v - 3x + 5y - 2z = -18 \\ 7u - 7v + 10x + 8y - 4z = 132 \\ -6u - 2v + 11x + 2y + 2z = -1 \end{cases}$$

## Вариант 4

$$\begin{cases} -4u + 8v - 8x + 2y + 7z = 148 \\ -4u - 7v + 4x + 4y - 6z = -159 \\ -2u + 4v + 2x + 3y - 3z = -9 \\ -8u - 6v - 6x + 2y + 8z = -8 \\ -3u + 5v + 3x - 3y - 7z = -70 \end{cases}$$

## Вариант 5

$$\begin{cases} -5u - 5v + 10x - 3y - 8z = 74 \\ -3u - 7v - 3x + 6y + 2z = -16 \\ 5u + 2v + 9x - 6y - 3z = 84 \\ 2u + 7v - 3x + 8y + 10z = -12 \\ 9u + 8v + 5x + 5y - 6z = 94 \end{cases}$$

## Вариант 6

$$\begin{cases} -5u + 11v - 3x + 8y + 5z = -69 \\ -3u + 10v - 2x - 6y + 7z = -11 \\ 11u + 11v + 6x + 9y - 2z = 113 \\ -8u - 5v + 2x + 9y + 9z = 0 \\ -4u + 2v - 8x + 3y - 2z = -125 \end{cases}$$

## Вариант 7

$$\begin{cases} -2u - 5v - 2x + 2y + 8z = 46 \\ 3u - 6v + 8x - 4y - 3z = -37 \\ -3u - 2v - 3x + 4y + 8z = 50 \\ -3u + 7v - 8x + 10y + 8z = 56 \\ 10u + 8v - 2x + 5y + 2z = 5 \end{cases}$$

## Вариант 8

$$\begin{cases} -8u + 2v + 8x - 3y - 6z = -23 \\ 7u + 7v - 5x - 7y + 7z = -59 \\ 5u - 4v + 2x - 3y - 8z = -66 \\ -3u + 9v - 4x - 5y - 4z = -54 \\ 8u + 8v + 6x + 7y - 6z = 79 \end{cases}$$

## Вариант 9

$$\begin{cases} -2u + 8v - 7x + 8y - 3z = 74 \\ 9u - 6v + 6x + 5y - 7z = -8 \\ -3u + 11v - 6x + 9y - 5z = 105 \\ 10u + 5v - 7x + 11y - 3z = 26 \\ 9u + 3v - 2x + 2y + 5z = -14 \end{cases}$$

## Вариант 10

$$\begin{cases} 3u + 2v - 4x + 9y + 6z = -128 \\ -3u + 6v - 2x - 2y - 5z = -24 \\ -7u - 3v - 8x - 2y + 8z = -118 \\ -4u + 2v + 8x - 8y - 3z = 107 \\ 6u - 4v - 2x - 2y + 10z = -32 \end{cases}$$

## Вариант 11

$$\begin{cases} 8u + 2v - 3x - 7y + 10z = 57 \\ 11u + 7v - 8x + 7y + 7z = -10 \\ 3u - 3v + 11x - 5y - 7z = 103 \\ -2u + 7v - 3x + 4y + 6z = -27 \\ -3u + 2v + 3x + 10y - 3z = -17 \end{cases}$$

## Вариант 12

$$\begin{cases} -4u + 3v + 11x - 2y + 7z = 69 \\ 6u - 2v + 8x + 9y + 4z = 101 \\ 7u + 11v - 3x - 6y - 5z = 55 \\ 11u - 3v - 5x - 8y + 2z = -18 \\ -3u + 6v + 2x + 4y - 5z = 25 \end{cases}$$

## Вариант 13

$$\begin{cases} 11u + 4v - 8x + 3y - 3z = 96 \\ -5u + 5v + 4x - 5y - 2z = -66 \\ 2u - 2v + 6x - 3y - 5z = -101 \\ -3u + 5v - 3x - 8y + 2z = -37 \\ -5u - 2v + 7x - 7y - 6z = -148 \end{cases}$$

## Вариант 14

$$\begin{cases} 3u + 8v + 6x - 5y - 2z = -3 \\ 2u - 4v + 8x - 3y - 3z = 62 \\ 11u + 10v + 5x + 11y + 8z = -61 \\ -7u + 11v - 3x - 6y + 8z = -36 \\ -5u - 2v - 6x + 2y - 6z = 9 \end{cases}$$

## Вариант 15

$$\begin{cases} 3u - 2v + 5x - 2y + 6z = 45 \\ 2u - 6v + 4x + 2y - 6z = -10 \\ -3u - 3v - 3x - 3y - 2z = -42 \\ -3u + 2v + 8x - 6y + 8z = 71 \\ -8u + 10v + 11x - 2y + 4z = 116 \end{cases}$$

## Вариант 16

$$\begin{cases} -8u + 2v + 2x + 7y + 7z = 70 \\ 2u + 11v + 8x - 3y - 8z = 109 \\ 9u - 3v + 3x + 8y + 10z = 110 \\ -8u - 2v + 2x + 10y + 8z = 32 \\ -8u - 6v - 8x + 3y + 7z = -88 \end{cases}$$

## Вариант 17

$$\begin{cases} 5u + 9v + 7x - 2y + 6z = 122 \\ 3u - 3v + 11x + 6y + 9z = 75 \\ 2u + 6v + 2x + 5y - 3z = 28 \\ -8u - 4v - 8x + 7y + 4z = -39 \\ 4u - 5v + 10x + 6y - 2z = -3 \end{cases}$$

## Вариант 18

$$\begin{cases} 5u - 4v - 3x - 3y + 2z = -79 \\ -2u - 3v - 5x + 6y + 5z = 8 \\ -2u + 2v - 3x - 2y + 10z = 31 \\ 3u + 7v + 10x + 2y + 6z = 229 \\ 6u - 5v - 8x + 2y + 3z = -77 \end{cases}$$

## Вариант 19

$$\begin{cases} -2u - 2v + 7x - 7y + 5z = -24 \\ -8u - 8v + 11x + 4y - 3z = 78 \\ 7u - 6v + 8x + 4y - 4z = 111 \\ -3u + 8v - 7x - 4y + 8z = -134 \\ 9u + 5v + 11x - 3y - 2z = 43 \end{cases}$$

## Вариант 20

$$\begin{cases} 5u - 8v - 3x - 2y - 5z = -97 \\ 9u - 3v + 11x - 6y + 8z = 111 \\ 11u + 3v + 2x + 11y + 10z = 91 \\ -5u + 2v + 7x - 4y - 6z = 16 \\ -7u + 9v - 6x - 3y - 3z = 44 \end{cases}$$

## Вариант 21

$$\begin{cases} -3u + 9v + 10x + 6y - 2z = 112 \\ 8u - 2v - 8x - 7y - 2z = -134 \\ -6u - 8v + 4x + 7y + 4z = 104 \\ 3u - 5v + 6x + 10y - 8z = 42 \\ -3u + 8v + 11x + 10y - 8z = 102 \end{cases}$$

## Вариант 22

$$\begin{cases} 11u + 6v + 6x - 4y + 2z = 103 \\ 11u + 9v + 6x + 11y + 9z = 71 \\ -3u + 6v + 10x + 8y - 4z = -31 \\ 4u + 3v + 10x + 8y + 4z = 65 \\ 9u + 8v + 6x + 3y - 3z = 38 \end{cases}$$

## Вариант 23

$$\begin{cases} 4u - 7v + 2x + 5y + 9z = 166 \\ 2u - 6v + 4x + 6y - 3z = 101 \\ 9u + 9v + 8x + 8y - 4z = 150 \\ -5u + 4v - 3x - 3y - 6z = -130 \\ 2u - 3v + 4x - 5y + 2z = 9 \end{cases}$$

## Вариант 24

$$\begin{cases} -5u + 2v + 5x - 3y - 5z = 64 \\ -8u - 5v + 4x + 7y + 4z = 45 \\ 9u + 2v + 4x + 2y + 11z = -99 \\ 11u + 6v + 2x + 5y - 7z = -92 \\ -7u + 2v + 3x + 2y - 2z = 29 \end{cases}$$

## Вариант 25

$$\begin{cases} 5u + 8v - 7x + 11y - 8z = -42 \\ 7u + 2v + 5x + 7y - 3z = -76 \\ 2u - 4v - 6x + 9y + 2z = -37 \\ -2u + 10v - 8x - 7y + 6z = 121 \\ -2u + 6v - 2x - 7y - 3z = 65 \end{cases}$$

## Вариант 26

$$\begin{cases} 4u + 3v + 6x + 10y + 7z = 17 \\ -7u + 4v + 11x - 2y - 3z = -30 \\ -3u + 2v - 3x + 2y + 4z = -32 \\ 4u + 6v + 9x + 2y - 6z = 22 \\ -6u - 6v + 8x + 7y + 9z = -9 \end{cases}$$

## Вариант 27

$$\begin{cases} 4u - 8v + 5x + 2y - 6z = -56 \\ 10u - 8v + 3x - 6y + 2z = 42 \\ -6u + 7v - 7x - 7y + 3z = 37 \\ 5u + 3v + 6x - 4y + 4z = -40 \\ 10u + 7v - 3x + 2y - 2z = 17 \end{cases}$$

### Вариант 28

$$\begin{cases} 10u - 5v - 3x + 11y - 3z = 161 \\ 4u - 8v - 3x + 7y + 6z = 121 \\ -5u - 4v + 4x - 8y + 4z = -43 \\ 10u - 6v - 2x - 5y + 7z = 135 \\ -8u + 2v + 7x + 2y + 3z = -60 \end{cases}$$

### Вариант 29

$$\begin{cases} 6u + 2v + 5x + 10y - 4z = 64 \\ -8u + 6v + 9x - 4y + 2z = 36 \\ -5u + 9v + 8x - 8y - 2z = -26 \\ -3u + 9v + 9x - 3y + 8z = 17 \\ -2u - 2v + 7x - 3y + 6z = 43 \end{cases}$$

### Вариант 30

$$\begin{cases} 5u + 2v + 2x + 7y + 6z = -70 \\ 11u - 4v + 4x + 5y - 3z = -56 \\ -7u + 2v + 10x - 6y + 8z = -56 \\ 2u - 3v - 8x - 3y + 2z = 65 \\ -2u + 5v + 8x + 3y + 2z = -83 \end{cases}$$

## Задание №2

### Вариант 1

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = 2\sin x \cos x + \sin x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \sin^2 x - \cos x.$$

### Вариант 2

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = 4\sin x \cos x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \sin^2 x - \cos^2 x.$$

## Вариант 3

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -e^{-x}\sin^2 x + 2e^{-x}\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = e^{-x}\sin^2 x.$$

## Вариант 4

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -2\sin x\cos x - \cos x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \cos^2 x - \sin x.$$

## Вариант 5

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -e^{-x} + 2\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = e^{-x} + \sin^2 x.$$

## Вариант 6

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = e^{x}\cos^2 x - 2e^{x}\sin x\cos x,\ a = 0,\ b = 2,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = e^{x}\cos^2 x.$$

## Вариант 7

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -2\sin x\cos x - \cos x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \cos^2 x - \sin x.$$

## Вариант 8

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = 2\sin x\cos x + \sin x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \sin^2 x - \cos x.$$

## Вариант 9

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -e^{-x} + 2\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = e^{-x} + \sin^2 x.$$

## Вариант 10

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = 4\sin x\cos x,\ a = 0,\ b = 1,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = \sin^2 x - \cos^2 x.$$

## Вариант 11

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = e^x\cos^2 x - 2e^x\sin x\cos x,\ a = 0,\ b = 2,$$

число разбиений $n = 10; 40; 160; 640$.

$$F(x) = e^x\cos^2 x.$$

## Вариант 12

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -e^{-x}\sin^2 x + 2e^{-x}\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^{-x}\sin^2 x.$$

## Вариант 13

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = -e^{-x}\sin^2 x + 2e^{-x}\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = e^{-x}\sin^2 x.$$

## Вариант 14

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = 2\sin x\cos x + \sin x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = \sin^2 x - \cos x.$$

## Вариант 15

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = -2\sin x\cos x - \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = \cos^2 x - \sin x.$$

## Вариант 16

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = e^{x}\cos^2 x - 2e^{x}\sin x\cos x,\ a = 0,\ b = 2,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = e^{x}\cos^2 x.$$

## Вариант 17

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = -e^{-x} + 2\sin x \cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = e^{-x} + \sin^2 x.$$

## Вариант 18

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом Симпсона.

$$f(x) = 4\sin x \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 20; 40; 80.

$$F(x) = \sin^2 x - \cos^2 x.$$

## Вариант 19

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = 2\sin x \cos x + \sin x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \sin^2 x - \cos x.$$

## Вариант 20

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = 4\sin x \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \sin^2 x - \cos^2 x.$$

## Вариант 21

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -e^{-x}\sin^2 x + 2e^{-x}\sin x \cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^{-x}\sin^2 x.$$

## Вариант 22

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -2\sin x\cos x - \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \cos^2 x - \sin x.$$

## Вариант 23

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = -e^{-x} + 2\sin x\cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^{-x} + \sin^2 x.$$

## Вариант 24

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом трапеций.

$$f(x) = e^x\cos^2 x - 2e^x\sin x\cos x,\ a = 0,\ b = 2,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^x\cos^2 x.$$

## Вариант 25

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -2\sin x\cos x - \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \cos^2 x - \sin x.$$

## Вариант 26

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = 2\sin x\cos x + \sin x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \sin^2 x - \cos x.$$

## Вариант 27

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -e^{-x} + 2\sin x \cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^{-x} + \sin^2 x.$$

## Вариант 28

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = 4\sin x \cos x,\ a = 0,\ b = 1,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = \sin^2 x - \cos^2 x.$$

## Вариант 29

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = e^x \cos^2 x - 2e^x \sin x \cos x,\ a = 0,\ b = 2,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^x \cos^2 x.$$

## Вариант 30

Вычислить определенный интеграл от функции $f(x)$ на промежутке $[a, b]$ методом прямоугольников.

$$f(x) = -e^{-x} \sin^2 x + 2e^{-x} \sin x \cos x,\ a = 0,\ b = 3,$$

число разбиений $n$ = 10; 40; 160; 640.

$$F(x) = e^{-x} \sin^2 x.$$

# Задание №3

## Вариант 1

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y + e^{-x} \cos x,\ y(a) = 0,\ a = 0,\ b = 2.$$

Точное решение задачи: $y(x) = e^{-x} \sin x$.

## Вариант 2

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y + \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x + \sin x$.

## Вариант 3

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y - e^x \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x \cos x$.

## Вариант 4

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y - e^{-x} \sin x,\ y(a) = 1,\ a = 0,\ b = 2.$$

Точное решение задачи: $y(x) = e^{-x} \cos x$.

## Вариант 5

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y + e^x \cos x,\ y(a) = 0,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x \sin x$.

## Вариант 6

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y + e^{-x},\ y(a) = 0,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = xe^{-x}$.

## Вариант 7

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y - e^x \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x \cos x$.

## Вариант 8

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y + \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x + \sin x$.

## Вариант 9

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y + e^x \cos x, y(a) = 0,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x \sin x$.

## Вариант 10

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y + e^{-x} \cos x,\ y(a) = 0,\ a = 0,\ b = 2.$$

Точное решение задачи: $y(x) = e^{-x} \sin x$.

## Вариант 11

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y + e^{-x},\ y(a) = 0,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = xe^{-x}$.

## Вариант 12

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y - e^{-x} \sin x,\ y(a) = 1,\ a = 0,\ b = 2.$$

Точное решение задачи: $y(x) = e^{-x} \cos x$.

## Вариант 13

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y - \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x + \cos x$.

## Вариант 14

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y - \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x + \cos x$.

## Вариант 15

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y - \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x - \sin x$.

## Вариант 16

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y - \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x - \sin x$.

## Вариант 17

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = y + \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x - \cos x$.

## Вариант 18

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = y + \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^x - \cos x$.

## Вариант 19

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y + \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \sin x$.

## Вариант 20

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y + \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \sin x$.

## Вариант 21

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y + \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \cos x$.

## Вариант 22

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y + \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \cos x$.

## Вариант 23

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y - \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \sin x$.

## Вариант 24

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y - \cos x - \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \sin x$.

## Вариант 25

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y - \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \cos x$.

## Вариант 26

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y - \cos x + \sin x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \cos x$.

## Вариант 27

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y + \cos^2 x - 2\sin x \cos x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \cos^2 x$.

## Вариант 28

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y + \cos^2 x - 2\sin x \cos x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} + \cos^2 x$.

## Вариант 29

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Эйлера.

$$f(x, y) = -y - \cos^2 x + 2\sin x \cos x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \cos^2 x$.

## Вариант 30

Решить задачу Коши для обыкновенного дифференциального уравнения $y' = f(x, y)$ на промежутке $[a, b]$ методом Рунге — Кутта четвертого порядка точности.

$$f(x, y) = -y - \cos^2 x + 2\sin x \cos x,\ y(a) = 1,\ a = 0,\ b = 1.$$

Точное решение задачи: $y(x) = e^{-x} - \cos^2 x$.

# Задание №4

## Вариант 1

Метод половинного деления

$$x + \ln(x + 0.5) - 0.5 = 0,\ x \in [0, 2].$$

## Вариант 2

Метод половинного деления

$$x^5 - x - 0.2 = 0,\ x \in [1,\ 1.1].$$

## Вариант 3

Метод половинного деления

$$x^4 + 2x^3 - x - 1 = 0,\ x \in [0,\ 1].$$

## Вариант 4

Метод половинного деления

$$x^3 - 0.2x^2 - 0.2x - 1.2 = 0,\ x \in [1,\ 1.5].$$

## Вариант 5

Метод половинного деления

$$2\sin^2\frac{x}{3} - 3\cos^2\frac{x}{4} = 0,\ x \in \left[0,\ \frac{\pi}{2}\right].$$

## Вариант 6

Метод половинного деления

$$x^4 + 0.8x^3 - 0.4x^2 - 1.4x - 1.2 = 0,\ x \in [-1.2,\ -0.5].$$

## Вариант 7

Метод половинного деления

$$x^4 - 4.1x^3 + x^2 - 5.1x + 4.1 = 0,\ x \in [3.7,\ 5.0].$$

## Вариант 8

Метод половинного деления

$$x \cdot 2^x - 1 = 0,\ x \in [0, 1].$$

## Вариант 9

Метод половинного деления

$$x^2 - \sin 5x = 0,\ x \in [0.5,\ 0.6].$$

## Вариант 10

Метод половинного деления

$$2\sin^2\frac{2x}{3} - 3\cos^2\frac{2x}{4} = 0,\ x \in \left[0,\ \frac{\pi}{4}\right].$$

## Вариант 11

Метод половинного деления

$$x^3 - 2x^2 + x - 3 = 0,\ x \in [2.1,\ 2.2].$$

## Вариант 12

Метод половинного деления

$$(4 + x^2)(e^x - e^{-x}) = 18,\ x \in [1.1,\ 1.3].$$

## Вариант 13

Метод половинного деления

$$x^4 + 0.5x^3 - 4x^2 - 3x - 0.5 = 0,\ x \in [-1.0,\ 0.5].$$

## Вариант 14

Метод половинного деления

$$x^2 - 1.3\ln(x + 0.5) - 2.8x + 1.15 = 0,\ x \in [2.0,\ 2.5].$$

## Вариант 15

Метод половинного деления

$$x^3 + x^2 - 3 = 0,\ x \in [0.6,\ 1.4].$$

## Вариант 16

Метод половинного деления

$$x^5 - x - 0.2 = 0,\ x \in [0.9,\ 1.1].$$

## Вариант 17

Метод половинного деления

$$5x^3 - x - 1 = 0,\ x \in [0.6,\ 0.8].$$

## Вариант 18

Метод половинного деления

$$x^3 - 2x - 5 = 0,\ x \in [1.9,\ 2.94].$$

## Вариант 19

Метод половинного деления

$$x^3 + x = 1000,\ x \in [9.1,\ 10.0].$$

## Вариант 20

Метод половинного деления

$$x^4 + 2x^3 - x - 1 = 0,\ x \in [0,\ 1.0].$$

## Вариант 21

Метод Ньютона

$$x^2 - \sin 5x = 0,\ x \in [0.5,\ 0.6].$$

## Вариант 22

Метод Ньютона

$$x^3 - 0.2x^2 - 0.2x - 1.2 = 0,\ x \in [1,\ 1.5].$$

## Вариант 23

Метод Ньютона

$$x^3 - 2x^2 + x - 3 = 0,\ x \in [2.1,\ 2.2].$$

## Вариант 24

Метод Ньютона

$$x^4 + 0.8x^3 - 0.4x^2 - 1.4x - 1.2 = 0,\ x \in [-1.2,\ -0.5].$$

## Вариант 25

Метод Ньютона

$$x^3 - 2x - 5 = 0,\ x \in [1.9,\ 2.94].$$

## Вариант 26

Метод секущих

$$x^4 + 0.8x^3 - 0.4x^2 - 1.4x - 1.2 = 0,\ x \in [-1.2,\ -0.5].$$

## Вариант 27

Метод секущих

$$x^2 - \sin 5x = 0,\ x \in [0.5,\ 0.6].$$

## Вариант 28

Метод секущих

$$x^3 - 0.2x^2 - 0.2x - 1.2 = 0,\ x \in [1,\ 1.5].$$

## Вариант 29

Метод секущих

$$x^3 - 2x^2 + x - 3 = 0,\ x \in [2.1,\ 2.2].$$

## Вариант 30

Метод секущих

$$x^3 - 2x - 5 = 0,\ x \in [1.9,\ 2.94].$$

# ЗАКЛЮЧЕНИЕ

В этом учебном пособии рассмотрены вопросы, касающиеся решения математических задач, наиболее часто встречающиеся в процессе моделирования электрических схем. При рассмотрении много внимания уделялось различным математическим формулировкам, да и описание всех алгоритмов весьма математизировано. У кого-то это вызовет уныние, но наличие математического описания это огромный плюс. Это говорит о том, что все поставленные задачи являются «жесткими», не имеющими расплывчатых формулировок, так как язык математики самый точный (в смысле описания) язык. Возможно у кого-то возникнет впечатление о черезмерной сложности использования этих методов для решения реальных задач, и что от них следует отказаться. Однако большинство реальных задач имеет хорошо отработанные методы и реализованные на их основе программы. Таким образом, усвоив алгоритм и реализовав программу, Вы в дальнейшем с легкостью будете применять ее для решения аналогичных задач, не задумываясь над тем, как это все работает. Конечно, при более глубоком изучении конкретной задачи и усложнении ее модели от Вас потребуются более сложные алгоритмы, но если Вы имеете под ногами мощный фундамент знаний, эта задача не будет выглядеть для Вас как нечто нереализуемое.

# ЛИТЕРАТУРА


[1] Практическое руководство по программированию / под ред. Б. Мик, П. Хит, Н. Рашби — М. : Радио и связь, 1986. — 186 с.

[2] Форсайт Дж. Машинные методы математических вычислений / Дж. Форсайт, М. Малькольм, К. Моулер. — М. : Мир, 1980. — 280 с.

[3] Каханер Д. Численные методы и программное обеспечение / Д. Каханер, К. Моулер, С. Нэш. — М. : Мир, 2001. — 575 с.

[4] Вирт Н. Систематическое программирование. Введение / Н. Вирт. — М. : Мир, 1977. — 184 с.

[5] Вирт Н. Алгоритмы + структуры данных = программы / Н. Вирт. — М. : Мир, 1985.

[6] Бахвалов Н. С. Численные методы / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельников. — 4-е изд. — М. : БИНОМ. Лаборатория знаний, 2006. — 636 с.

[7] Фармалеев В. Ф. Численные методы / В. Ф. Фармалеев, Д. Л. Ревизников. — 2-е изд., испр., доп. — М. : ФИЗМАТЛИТ, 2006. — 400 с.

[8] Киреев В. И. Численные методы в примерах и задачах : учеб. пособие / В. И. Киреев, А. В. Пантелеев. — 2-е изд., стер. — М. : Высш. шк., 2006. — 480 с.

# ГЛОССАРИЙ

*Абстракция* — форма познания, основанная на мысленном выделении существенных свойств и связей предмета и отвлечении от других, частных его свойств и связей.

*Алгоритм* — точное, общепринятое предписание, определяющее процесс преобразования исходных данных в искомый результат.

*Аппроксимация* — научный метод, состоящий в замене одних объектов другими, в том или ином смысле близкими к исходным, но более простыми.

*Ведущий элемент* (матрицы) — элемент строки или столбца, относительно которого в методе Гаусса осуществляются вычисления для исключения соответствующей ему переменной в других строках или столбцах.

*Выполнимость алгоритма* (или массовость) — возможность исходить из любых исходных данных, принадлежащих некоторому множеству $G$ исходных данных.

*Гладкая функция* — функция, имеющая непрерывную производную на всём множестве определения.

*Декомпозиция* — научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых.

*Дифференциальное уравнение* — уравнение, связывающее независимую переменную, искомую функцию и ее производную.

*Дифференциальное уравнение в частных производных* — дифференциальное уравнение, в котором неизвестная функция зависит от нескольких переменных.

*«Жесткие уравнения»* — дифференциальные уравнения, решение которых получить намного проще с помощью определенных неявных методов, чем с помощью явных методов.

*Задача Коши* — формулировка задачи нахождения решения дифференциального уравнения с условиями, заданными в одной точке.

*Интерполяция* — способ нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений.

*Квадратурная формула* — формула численного интегрирования, обеспечивающая приближенное равенство $\int_a^b f(x)\,dx \approx \sum_{k=0}^{n} C_k y_k$.

*Конечность алгоритма* (или результативность) — свойство определять процесс, который для любых допустимых исходных данных приводит к получению искомого результата.

*Конечно-разностные методы* — методы в которых конечная система алгебраических уравнений ставится в соответствие какой-либо дифференциальной задаче.

*Кубический сплайн* — интерполяционная функция $S(x)$, удовлетворяющая условиям: на каждом отрезке $[x_{k-1}, x_k]$, $k = 1, 2, \ldots, n$ функция $S(x)$ является полиномом 3-ей степени; функция $S(x)$, а также ее первая и вторая производные непрерывны на $[a, b]$; $S(x_k) = f(x_k)$, $k = 0, 1, \ldots, n$.

*Кусочно-полиномиальные функции* (сплайн) — функция, определенная на отрезке и имеющая на этом отрезке некоторое число непрерывных производных.

*Лагранжева интерполяция* — интерполяция полиномами Лагранжа.

*Ленточные матрицы* — матрицы, все ненулевые элементы которых расположены вблизи главной диагонали.

*Линейная алгебра* — часть алгебры, изучающая векторы, векторные или линейные пространства, линейные отображения и системы линейных уравнений.

*Линейная комбинация* — конечная сумма вида: $\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n$.

*Математическая модель* — модель объекта, отражающая в математической форме важнейшие его свойства — законы, которым он подчиняется, связи, присущие составляющим его частям, и т. д.

*Машинная точность* — разница между 1.0 и следующим по величине вещественным числом.

*Метод Адамса* (решения ДУ) — метод, позволяющий найти решение с использованием известных решений в нескольких соседних точках.

*Метод восходящего проектирования* (или проектирование снизу вверх) — сначала изучается имеющаяся вычислительная система, затем собираются последовательности инструкций в элементарные процедуры, типичные для решаемой задачи. Элементарные процедуры затем используются на следующем, более высоком уровне иерархии процедур.

*Метод Гаусса* (решения СЛУ) — метод последовательного исключения переменных, когда с помощью элементарных преобразований система уравнений приводится к равносильной системе ступенчатого (или по другому — треугольного) вида, из которой последовательно, начиная с последних (по номеру) переменных, находятся все остальные переменные.

*Метод Зейделя* (решения СЛУ) — итерационный метод в котором при вычислении $(k + 1)$-ого приближения ранее полученные приближения сразу же используются в вычислениях.

*Метод итераций* (решения частичной проблемы собственных значений) — метод, позволяющий приближенно (с заданной точностью ε) получить собственные значения матрицы $A$, имеющей $n$ линейно независимых собственных векторов $X_i$, $1 \leqslant i \leqslant n$.

*Метод непосредственного развертывания* (решения полной проблемы собственных значений) — метод нахождения всех собственных значений матрицы, сводящийся к задаче нахождения корней полинома.

*Метод нисходящего проектирования* (или проектирование сверху вниз) — поэтапная декомпозиция, сочетающаяся с одновременной детализацией программы.

*Метод Ньютона* (решения трансцендентных уравнений) — метод нахождения корня нелинейного уравнения вида $f(x) = 0$, основанный на последовательном приближении к решению посредством касательных. Обладает высокой степенью сходимости, но требует вычисления первой производной функции.

*Метод половинного деления* (решения трансцендентных уравнений) — метод нахождения корня нелинейного уравнения, основанный на уменьшении интервала неопределенности в два раза. Обладает низкой скоростью сходимости.

*Метод пошаговой детализации* (разработки программ) — метод, при котором, вначале разрабатывается общая структура программы, состоящая из записи алгоритма более крупными блоками, на следующих этапах каждый из этих блоков детализируется на более мелкие, пока в результате не получатся элементарные действия.

*Метод прогонки* (решения СЛУ) — модифицированный метод Гаусса, учитывающий структуру ленточной матрицы.

*Метод простой итерации* (решения СЛУ) — итерационный метод, позволяющий находить решение СЛУ, не храня в памяти всю матрицу коэффициентов.

*Методы Рунге — Кутта* (решения ДУ) — одношаговые методы решения ДУ, не требующие вычисления производных.

*Метод секущих* (решения трансцендентных уравнений) — метод нахождения корня нелинейного уравнения вида $f(x) = 0$, основанный на последовательном приближении к решению посредством секущих. Обладает высокой степенью сходимости, не требует вычисления первой производной функции.

*Метод Эйлера* (решения ДУ) — одношаговый метод, вычисляющий следующее значение посредством прямолинейной экстраполяции из предыдущей точки.

*Начальное приближение* — некоторое начальное значение искомого значения для итерационных методов.

*Невырожденная система линейных уравнений* — система линейных уравнений, для которой определитель матрицы коэффициентов отличен от 0.

*Невязка* — ошибка (погрешность) в результате вычислений $(r = \mathbf{b} - A\mathbf{x}_*)$.

*Неявная разностная схема* — конечная система алгебраических уравнений, поставленная в соответствие какой-либо дифференциальной задаче, использует урав-

нения, которые выражают данные через несколько соседних точек результата, обычно являются устойчивыми.

*Норма* — число, которое измеряет общий уровень элементов вектора.

*Обратная матрица* — матрица, при умножении исходной на которую получается единичная матрица.

*Определенность алгоритма* — точность, не оставляющая места для произвола, и общепонятность.

*Ошибка* (решения) — разница между точным и приближенным решением ($e = \mathbf{x} - \mathbf{x}_*$).

*Ошибка дискретизации* — разность между вычисленным решением и теоретическим решением, при условии что отсутствуют ошибки округления.

*Ошибка округления* — ошибка, возникающая вследствие ограниченности вещественных чисел в представлении компьютера.

*Плохо обусловленная матрица* — матрица, определитель которой очень близок к 0, но не равен 0.

*Полином* — многочлен вида: $c_0 + c_1x^1 + c_2x^2 + \ldots + c_nx^n$.

*Полином Лагранжа* — полином вида $l_j(x) = \prod_{i=0,\, i\neq j}^{n} \frac{(x - x_i)}{(x_j - x_i)}$.

*Полиномиальная интерполяция* — интерполяция полиномами.

*Полная проблема собственных значений* — задача нахождения всех собственных значений.

*Прямолинейная экстраполяция* — аппроксимация полиномами первой степени, при котором функция аппроксимируется вне заданного интервала.

*Псевдокод* — некоторый не слишком формализованный язык, промежуточный между естественным языком и языком программирования.

*Равномерная разностная сетка* — разностная сетка с одинаковым расстоянием между прямыми.

*Разностная сетка* — параллельные прямые, «покрывающие» области определения и значений.

*Разреженная матрица* — матрица, большинство элементов которой нули.

*Рекуррентные алгоритмы* — алгоритмы, построенные на рекуррентных соотношениях.

*Рекуррентные соотношения* — выражения вида $v_i = f(v_{i-1})$ для всех $i > 0$.

*Рекурсия* — возникает, если рассматриваемый объект содержит сам себя или определен с помощью самого себя.

*Сеточная функция* — функция, определенная в узлах разностной сетки.

*Собственное значение матрицы* — собственными значениями действительной квадратной матрицы $A$ называют числа $\lambda$, в общем случае комплексные, при которых определитель матрицы: $|A - \lambda \cdot E| = 0$.

*Собственный вектор матрицы* — собственным вектором матрицы $A$, соответствующим собственному числу $\lambda_i$, называют вектор $t_i$, для которого справедливо соотношение: $A \cdot t_i = \lambda_i \cdot t_i$ , где $i = 1...n$.

*Сплайн-интерполяция* — то же, что и полиномиальная интерполяция.

*Структура данных* — абстракция реальных объектов, сформулированная в терминах конкретного языка программирования.

*Структурное программирование* — программирование без GOTO.

*Трехдиагональная матрица* — матрица, у которой отличные от 0 элементы расположены только на главной диагонали и на диагоналях, расположенных непосредственно над и под главной диагональю.

*Узлы разностной сетки* — точки пересечения прямых, образующих разностную сетку.

*Устойчивость решения* — означает, что решения, при достаточно близких начальных значениях будут «не сильно» отличаться друг от друга.

*Формула «левых» прямоугольников* — аппроксимация подынтегральной функции полиномом нулевой степени с выбором в качестве аргумента левой границы отрезка.

*Формула «правых» прямоугольников* — аппроксимация подынтегральной функции полиномом нулевой степени с выбором в качестве аргумента правой границы отрезка.

*Формула Симпсона* — аппроксимация подынтегральной функции полиномом второй степени, построенным по трем узлам: $x_{i-1}$, $x_{i-1/2}$, $x_i$.

*Формула трапеций* — аппроксимация подынтегральной функции полиномом первой степени, построенном по двум узлам: $x_i$ и $x_{i+1}$.

*Формула «центральных» прямоугольников* — аппроксимация подынтегральной функции (т. е. приближенная замена ее) полиномом нулевой степени с выбором в качестве аргумента середины отрезка.

*Хранимая матрица* — матрица, все $n^2$ элементов которой хранятся в оперативной памяти компьютера.

*Частичная проблема собственных значений* — задача нахождения только некоторых собственных значений.

*Численные алгоритмы* — алгоритмы решения математических задач в численном виде.

*Число обусловленности матрицы* — мера того, насколько хорошо или плохо обусловлена матрица.

*Число с плавающей точкой* — форма представления действительных чисел, в которой число хранится в форме мантиссы и показателя степени.

*Шаг разностной схемы* — расстояние между прямыми, образующими разностную сетку.

*Эрмитова интерполяция* — метод полиномиальной интерполяции, в котором строится многочлен, значения которого в выбранных точках совпадают со значениями исходной функции в этих точках, и производные многочлена в данных точках совпадают со значениями производных функции.

*Явная разностная схема* — конечная система алгебраических уравнений, поставленная в соответствие какой-либо дифференциальной задаче, использует уравнения, которые выражают данные через несколько точек соседних точек результата, полученного на предыдущем шаге, часто оказываются неустойчивыми.

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Учебное издание

**Мещеряков** Павел Сергеевич

ПРИКЛАДНАЯ ИНФОРМАТИКА

Учебное пособие

Корректор Осипова Е. А.
Компьютерная верстка Лигай Т. А.

Подписано в печать 12.10.12. Формат 60x84/8.
Усл. печ. л. 15,35. Тираж 300 экз. Заказ

---

Издано в ООО «Эль Контент»
634029, г. Томск, ул. Кузнецова д. 11 оф. 17
Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.