ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МОЛДОВЫ

Центр Образования и Исследований

в Математике и Информатике

MRDA/CRDF

Теоретическая и Прикладная Математика

Monograph Series

БОРИС РЫБАКИН

ЧИСЛЕННЫЕ МЕТОДЫ ДЛЯ МНОГОПРОЦЕССОРНЫХ ЭВМ

Кишинев * 2008

CZU 519.6
R ??

В данной книге изложены численные методы решения задач по специальности "Прикладная математика". Кроме традиционного курса лекций в книге приводится информация по основам математического моделирования, по языку программирования Фортран, а также параллельные алгоритмы численного решения задач математической физики. Главным стимулом написания этой книги было желание расширить круг людей – студентов, аспирантов, преподавателей, научных сотрудников, которые хотят профессионально заниматься исследованиями в области математического моделирования и численных методов. Как показывает опыт, в современных условиях недостаточно владеть теоретическими основами численных методов. Рост быстродействия компьютеров, появление доступных по цене двух, четырех и более ядерных процессоров с современными операционными системами, языками высокого уровня с встроенными операторами распараллеливания (Фортран 90, Фортран 95) и поддерживающими многопроцессорность – все это заставляет модернизировать традиционные курсы по численным методам. Данная монография содержит курс лекций, в котором учтен современный уровень программного обеспечения и уровень развития вычислительной техники.

Автор выражает благодарность доктору ф.-м.н. Секриеру И.В. за ценные предложения и замечания.

Автор: *Рыбакин Б.П., Доктор хабилитат*
Рецензент: *Секриеру И.В., Доктор, конференциар*

Recommended for publication by
Scientific Council of Center for Education and Research
in Mathematics and Computer Science

Descrierea CIP a Camerei Naţionale a Cărţii

Борис Рыбакин

Numerical methods for multiprocessor computer. Moldova State University, Center for Education and Research in Mathematics and Computer Science (MRDA/CRDF) – Ch.: CEP USM, 2008. – 337 p.

Bibliogr. p. 335 (36 tit.)

ISBN 978-9975-70-691-9

150 ex.

519.6

ISBN 978-9975-70-691-9

Оглавление

1 Предисловие **7**

1.1. **Для чего нужна эта книга** 7

1.2. **Кому нужна эта книга** 10

1.3. **Математическое моделирование** 11

1.4. **Конструирование программ** 14

1.5. **Выбор языка программирования** 16

1.5.1. **Тестирование кода и его надежность** 18

1.5.2. **Преимущество Фортрана при создании вычислительных программ** 18

2 Ошибки вычислений **19**

2.1. **Введение** 19

2.2. **Ошибки вычислений** 19

2.2.1. **Распространение ошибки** 21

2.3. **Представление целых чисел** 22

2.4. **Представление вещественных чисел** 23

2.5. **Ошибки в научных вычислениях** 27

2.6. **Отладка и тестирование программ** 30

2.7. **Упражнения** 34

3 Решение систем линейных уравнений **36**

3.1. **Введение** 36

3.2. **Метод простой итерации** 40

3.2.1. **Итерации Гаусса – Зейделя** 41

3.3. **Метод Гаусса** 42

3.3.1. **Плохая обусловленность систем линейных уравнений** 45

3.3.2. **Программа решения системы линейных уравнений методом исключения Гаусса** 47

3.4. **Метод прогонки** 51
3.5. **Упражнения** 56

4 Решение нелинейных уравнений **58**
4.1. **Введение** 58
4.2. **Решение нелинейных уравнений** 59
4.2.1. **Метод простой итерации** 63
4.2.2. **Метод дихотомии (деления отрезка пополам)** 65
4.2.3. **Метод Ньютона** 68
4.2.4. **Метод секущих (хорд)** 72
4.2.5. **Метод Мюллера** 74
4.3. **Решение систем нелинейных уравнений** 78
4.3.1. **Метод простой итерации** 78
4.3.2. **Метод Зейделя** 80
4.3.3. **Метод Ньютона для решения систем нелинейных уравнений** 81
4.4. **Упражнения** 88

5 Численные методы оптимизации **91**
5.1. **Введение** 91
5.2. **Методы отыскания безусловного экстремума** 94
5.2.1. **Метод золотого сечения** 94
5.2.2. **Метод Ньютона** 97
5.3. **Функции нескольких переменных** 103
5.3.1. **Метод покоординатного спуска** 105
5.3.2. **Градиентные методы** 109
5.3.3. **Метод наискорейшего спуска** 110
5.4. **Упражнения** 114

6 Приближение функций **115**
6.1. **Введение** 115
6.2. **Интерполяционный полином Лагранжа** 119
6.2.1. **Ошибки интерполяции** 123
6.3. **Интерполяционный полином Ньютона** 124
6.4. **Интерполяционные полиномы Чебышёва** 132
6.5. **Интерполяция сплайнами** 136
6.6. **Метод наименьших квадратов** 140
6.7. **Упражнения** 142

7 Численное дифференциирование и интегрирование 144
7.1. **Введение** . . . 144
7.2. **Конечные разности** . . . 146
7.2.1. **Дифференциирование полинома Лагранжа** . . . 148
7.2.2. **Дифференциирование полинома Ньютона** . . . 149
7.3. **Численное интегрирование** . . . 151
7.3.1. **Кратные интегралы** . . . 158
7.3.2. **Оценка погрешности** . . . 159
7.4. **Упражнения** . . . 160

8 Численные методы решения ОДУ 162
8.1. **Введение** . . . 162
8.2. **Метод Эйлера** . . . 166
8.2.1. **Граничные условия** . . . 167
8.2.2. **Точность метода Эйлера** . . . 170
8.3. **Модифицированный метод Эйлера** . . . 170
8.3.1. **Метод Гюна** . . . 174
8.4. **Методы Рунге – Кутта** . . . 178
8.4.1. **Модификация метода Рунге – Кутта** . . . 181
8.5. **Многошаговые методы** . . . 184
8.5.1. **Метод Адамса** . . . 185
8.5.2. **Метод Милна – Симпсона** . . . 187
8.5.3. **Метод Хемминга** . . . 188
8.6. **Неявные методы** . . . 190
8.7. **Решение систем ОДУ** . . . 193
8.8. **Упражнения** . . . 199

9 Численные методы решения ДУЧП 202
9.1. **Введение** . . . 202
9.2. **Классификация дифференциальных уравнений** . . . 203
9.2.1. **Понятие корректно поставленной задачи** . . . 206
9.2.2. **Начальные и граничные условия для уравнений в частных производных** . . . 208
9.3. **Метод сеток** . . . 209
9.3.1. **Дискретизация производных** . . . 210
9.3.2. **Точность процесса дискретизации** . . . 211
9.4. **Уравнение переноса** . . . 213
9.4.1. **Схема Лакса** . . . 214
9.4.2. **Схема Лакса – Вендроффа.** . . . 215

9.4.3. **Схема "крест"** 216
9.4.4. **Схема "кабаре".** 220
9.5. **TVD и ENO схемы** 221
9.5.1. **Метод Бориса – Бука (коррекции потоков)** . . 222
9.5.2. **Схема TVD (Total Variation Diminition)** . . . 223
9.5.3. **Схема ENO (Essentialy Non Oscillatory)** 229
9.6. **Гиперболические уравнения.** 230
9.6.1. **Волновое уравнение** 230
9.6.2. **Начальные и граничные условия** 232
9.7. **Параболические уравнения** 235
9.7.1. **Схемы Ричардсона и Дюфорта – Франкела** . . 238
9.7.2. **Неявная схема Кранка – Николсона** 240
9.7.3. **Нелинейные уравнения** 244
9.8. **Эллиптические уравнения** 246
9.8.1. **Граничные условия** 247
9.8.2. **Метод последовательной сверхрелаксации (SOR)** 250
9.9. **Упражнения** 257

10 Технологии параллельного программирования 258
10.1. **Введение** 258
10.2. **Архитектура параллельных ЭВМ** 259
10.2.1. **Топология вычислительных систем** 264
10.3. **Алгоритмы параллельного программирования** 265
10.3.1. **Планирование вычислений** 268
10.4. **Технология параллельного программирования** 271
10.4.1. **Языки и методы параллельного программирования** 274
10.4.2. **OpenMP** 275
10.4.3. **MPI** 299

11 Введение в параллельные численные методы 308
11.1. **Введение** 308
11.2. **Теоретические основы параллельных методов** 309
11.3. **Параллельные алгоритмы векторно - матричных операций** 313
11.4. **Параллельные алгоритмы решения ДУЧП** 322
11.4.1. **Решение эллиптических уравнений** 322

Глава 1

Предисловие

1.1. Для чего нужна эта книга

В данной монографии изложены численные методы решения задач по специальности "Прикладная математика". Монография написана на основе курса лекций по "Численным методам", которые автор читает в Приднестровском государственном университете на физико – математическом факультете по специальностям "Прикладная математика" и "Информатика"и на спецкурсах для студентов старших курсов в Молдавском государственном университете. Кроме традиционного курса лекций в книге приводится информация по основам математического моделирования, по языку программирования Фортран, а также параллельные алгоритмы численного решения задач математической физики.

Главным стимулом написания этой книги было желание расширить круг людей – студентов, аспирантов, преподавателей, научных сотрудников, которые хотят профессионально заниматься исследованиями в области математического моделирования и численных методов. Как показывает опыт, в современных условиях недостаточно владеть теоретическими основами численных методов. Очень часто у студентов и аспирантов возникает психологический барьер перед практическим использованием численных методов, особенно с использованием современных языков программирования для многопроцессорных систем. И если теоретический материал усваивается достаточно успешно, то переход к программной реализации полученных знаний (на языках высокого уровня Фортран 90 и C++) представляет собой серьезную проблему. Существует определенный круг книг [1], [2] которые в той или иной степени восполняют этот пробел, но с нашей точки зрения они либо не совсем соответствуют

программе курса, либо несколько устарели.

Рост быстродействия компьютеров, появление доступных по цене двух, четырех и более ядерных процессоров с современными операционными системами, языками высокого уровня с встроенными операторами распараллеливания (Фортран 90, Фортран 95) и поддерживающими многопроцессорность – все это заставляет модернизировать традиционные курсы по численным методам.

Параллельно с этим развиваются и совершенствуются и сами численные методы. В связи с этим заметно некоторое отставание в преподавании численных методов, которые традиционно читаются по специальности прикладная математика, а также физикам, химикам, биологам, инженерам - по сравнению с возможностями современных компьютеров и программных средств. Таким образом, уже на этапе обучения закладывается отставание в знаниях и в практической квалификации выпускаемых специалистов.

С нашей точки зрения, необходим курс лекций, в котором учтен современный уровень программного обеспечения и уровень развития вычислительной техники. Кроме того, по мере развития методов математического моделирования, все более актуальными и востребованными становятся методы решения дифференциальных уравнений, как обыкновенных, так и, в особенности, в частных производных, которые применяются при решении задач гидроаэродинамики, механики деформируемого твердого тела и т.д.

Данная монография поэтому содержит некоторую дополнительную информацию, которая отсутствует в государственном стандарте (и которую можно опустить). Так, в Главах 10 и 11 приводятся начальные сведения по технологиям параллельного программирования – OpenMP и MPI, необходимые для построения параллельных программ. Эти сведения позволят осуществить параллельную обработку информации на многопроцессорных вычислительных системах. Большое внимание в книге уделено графической обработке полученных результатов с использованием встроенного в Фортран графического пакета Array Visualiser.

Структура монографии: **Первая глава** содержит сведения об основных понятиях математического моделирования, конструировании программ и необходимую информацию об языках программирования. Приведены нужные сведения о графическом пакете Array Visualiser, позволяющим наглядно представлять результаты численных расчетов.

Во **второй главе** содержится информация об ошибках вычислений. Ошибками вычислений называются не только ошибки, связанные с точ-

ностью представления чисел в ЭВМ, не только потерю точности при вычислениях и округлениях, но и логические ошибки, а также возможности их устранения. Как показывает опыт практической работы, изучению приемов отладки и тестирования написанных программ не уделяется должного внимания. Но при оценке времени, затраченного на получение решения той или иной задачи, можно отметить, что потери времени на нахождение логических ошибок составляет более 70 процентов общего времени, затраченного на написание и отладку программы.

Третья глава посвящена решению систем линейных уравнений. Рассматриваются традиционные методы – простой итерации, Гаусса, прогонки для решения ленточных систем уравнений. Метод прогонки будет в дальнейшем использоваться для решения обыкновенных дифференциальных уравнений и дифференциальных уравнений в частных производных неявными методами.

В **четвертой главе** излагается материал по методам решения нелинейных уравнений и систем нелинейных уравнений. Приведены методы дихотомии, Ньютона, секущих, метод Мюллера. Приведены тексты программ этих методов.

В **пятой главе** приведены необходимые сведения о численных методах оптимизации. Изложены данные об отыскании экстремума функции одного переменного, а также методы отыскания экстремума функции многих переменных.

Шестая глава посвящена изучению численных методов интерполяции и экстраполяции. Подробно рассматриваются интерполяционные полиномы Лангранжа, Ньютона, Чебышёва, аппроксимация сплайнами.

В **седьмой главе** изложены численные методы дифференциирования и интегрирования, методы дифференциирования интерполяционных полиномов Лагранжа и Ньютона. Обсуждаются различные методы вычисления определенных интегралов: методы трапеций, Симпсона, Буля.

В **восьмой главе** представлены численные методы решения обыкновенных дифференциальных уравнений (ОДУ). Изложены методы Эйлера, Гюна, Рунге – Кутта, Рунге – Кутта – Фехлберга, Адамса, Милна – Симпсона, Хэмминга, а также неявные методы решения систем ОДУ.

В **девятой главе** приводятся численные методы решения уравнений в частных производных. Даются основные понятия метода сеток, основные понятия явных и неявных разностных схем. Подробно изложены методы решения уравнения переноса. Приведены методы решения гиперболических, параболических и эллиптических уравнений.

В **десятой главе** излагается материал по основным технологиям па-

раллельного программирования OpenMP и MPI.

Одиннадцатая глава посвящена основам построения параллельных алгоритмов для методов, изученных в первых главах.

1.2. Кому нужна эта книга

Эта монография предназначен тем, кто собирается изучить численные методы и научиться получать решения поставленных задач с помощью языка программирования Fortran 90. Книга также будет полезна научным работникам, преподавателям, аспирантам и студентам, а также всем, кто использует численные методы в своей работе. Материал, изложенный в книге, можно использовать и для организации курсов с другой продолжительностью – преподаватель может выбрать темы своего курса сообразно его интересам.

Материал излагается таким образом, чтобы читателю было понятно как применяются и работают численные методы и какие ограничения они накладывают. В книге используются основы математического анализа, приводится математическое обоснование приведенных методов. Приведены необходимые сведения из курса математического анализа, линейной алгебы, теории дифференциальных уравнений и других дисциплин. Приведены небольшие упражнения, которые можно выполнить без использования компьютера.

Новые концепции. Изучение численных методов без создания работоспособных программ, которые реализуют эти методы, не позволяет овладеть практическими навыками использования изучаемых дисциплин. В этой книге мы стремились соблюсти баланс между большим количеством подробностей и ясностью, логичностью и полнотой изложения как численных методов, так и языка программирования.

Последовательность изложения. В этой книге мы начинали с изложения самых простых методов и примерах, построенных на этих методах, заканчивая полноценными программами для численного решения задач механики сплошной среды с использованием многопроцессорных систем. Интенсивность подачи материала рассчитана таким образом, чтобы читатель мог без особой спешки усваивать пройденный материал. В конце каждой главы приведены контрольные вопросы и упражнения, что позволяет использовать эту книгу в качестве учебного пособия для самостоятельной работы студентов. Для облегчения усвоения материала мы постарались снабдить его максимальным количеством графического

материала.

Знания, необходимые для чтения этой книги. Материал, который приводится в этой книге, доступен тем, кто начинает изучение численных методов. Однако наличие опыта программирования на любом языке программирования будет нелишним. Желательно, чтобы читатель этой книги имел представление об компиляции и отладке программ.

Все примеры, приведенные в данной книге, были написаны и отлажены с помощью языка программирования Fortran 90 в операционных системах MS Windows XP, MS Windows Vista, Sun Solaris 10. Были использованы оболочки MS Visual Studio, MS Visual Studio 2005, Sun Studio 12. Для работы с программами потребуется компилятор Fortran 90. Для корректной работы компиляторов рабочий компьютер должен обладать достаточным объемом дискового пространства, оперативной памяти и хорошим быстродействием.

Упражнения. Каждая глава содержит упражнения и контрольные вопросы которые предполагают создание законченных программ, их отладку и тестирование на Visual Fortran 90. Упражнения рассчитаны на самостоятельное выполнение в качестве лабораторных и самостоятельных работ.

1.3. Математическое моделирование и вычислительный эксперимент.

В последнее время обозначился большой интерес к численным методам в связи со все большим распространением методов математического моделирования. Математическая модель, это, в той или иной степени приближенное описание физических, химических, биологических и других явлений или объектов реального мира на языке математических уравнений. Основной целью математического моделирования является изучение объектов и возможность предсказания их поведения в различных условиях. Очень часто провести реальный эксперимент либо невозможно, либо такой эксперимент может нанести неприемлемо большой вред окружающей среде, либо он очень дорог. В таких случаях математическое моделирование и связанный с ним численный эксперимент оказывается незаменимым средством получения достоверной информации об изучаемом объекте.

Методология математического и численного моделирования получила свое обоснование и развитие в трудах А.А.Самарского [3, 4, 5]. Он пред-

ложил ввести следующие этапы математического моделирования механических, физических, биологических и других явлений:

1. Осмысление явления, подлежащего моделированию.

2. Описание исследуемого явления с помощью систем математических уравнений – создание математической модели исследуемого явления.

3. Создание или нахождение метода решения.

4. Формализация найденного метода решения при помощи какого - либо языка программирования.

5. Отладка полученной программы.

6. Тестирование программы, сравнение с аналитическими решениями, экспериментальными данными; при необходимости введение изменений в математическую модель и повторение этапов 3 – 6.

7. Проведение вычислительного эксперимента.

В последнее время математические модели становятся одним из основных инструментов познания явлений окружающего мира. Чаще всего, в качестве математического аппарата используются либо обыкновенные дифференциальные уравнения либо дифференциальные уравнения в частных производных.

Для решения таких систем дифференциальных уравнений в основном используют численные методы. Поэтому понятие численное моделирование применяется практически также часто, как и математическое моделирование, хотя эти термины не совсем эквивалентны. На самом деле, строго говоря, численная модель – это математическая модель + численный метод решения + программа.

Для того, чтобы дать количественно правильные предсказания, модель должна описывать как отдельные явления, происходящие в исследуемой системе, так и взаимодействие этих явлений. После создания математической модели и ее численного эквивалента, после отладки и тестирования построенную модель можно использовать для распространения теоретических предположений на область новых режимов и параметров. Численная модель часто оказывается полезной при инженерном проектировании и количественной проверке различных теорий.

Одним из важных применений численных моделей является проверка адекватности и полноты нашего понимания физических, химических биологических и других систем путем сравнения результатов расчетов с экспериментальными данными. Если математическая и численная модели точны, то при исследованиии результатов численных расчетов можно найти новые эффекты.

Совместное использование фундаментальных результатов, математического и численного моделирования и экспериментов при изучении сложных систем и явлений, представляет собой относительно новый подход. Экспериментальные исследования и математические модели помогают предложить законы, которым должна подчиняться изучаемая система.

Методы их изучения путем анализа результатов решения на ЭВМ соответствующих математических моделей, занимает промежуточное положение между экспериментами и теоретическими методами. С одной стороны, проведение каждого расчета на ЭВМ похоже на проведение физического эксперимента: исследователь "включает" уравнения, задает начальные и граничные условия, а затем следит за тем, что происходит; именно это же самое делает и экспериментатор. С другой стороны, получение численного решения не связано с физическим измерением параметров изучаемого явления и правдоподобность полученных результатов прямо связана с достоверностью математических моделей, которые применяются для описания изучаемых явлений.

Развитие вычислительной техники и методов численного решения задач дает возможность исследовать все более сложные нелинейные явления и процессы. Большой прогресс в уменьшении стоимости численного эксперимента достигнут на этапе разработки методов и алгоритмов решения задач. Так, согласно некоторым данным, стоимость решения на ЭВМ задач, основанных на двумерных уравнениях Навье – Стокса уменьшилось почти в 1000 раз [6].

Существенное значение имеет уменьшение затрат на разработку и модификацию программ. При этом очень большое значение имеет квалификация разработчика модели и кодировщика программ. Поэтому создание курса лекций по современным численным методам и по языкам высокого уровня позволяет подготавливать специалистов, способных грамотно и эффективно разрабатывать математические и численные модели.

Большое значение имеет и комплексный, структурный подход к созданию программ. К сожалению, в большинстве книг по численным методам отстутствует важная часть, посвященная созданию эффектив-

ных программных модулей, их отладке и тестированию. Разработка программ с позиции структурного подхода, должна вестись "сверху вниз" методом пошаговой детализации: в начале необходимо описать алгоритм решения задачи в целом, а затем необходимо детализировать каждое выделенное действие. Этот процесс аналогичен обычному анализу и декомпозиции действий, применяемом математиками при решении задач. При таком подходе на каждом этапе разработки программы разработчик имеет возможность сосредоточить свои усилия на отдельных вопросах, и отложить дальнейшую детализацию на следующий этап разработки.

1.4. Конструирование программ

Практика показывает, что при создании программ неизбежно появляются ошибки. Этап проектирования сложных программ, предназначенных, в частности, для построения математических моделей и проведения по ним вычислительного эксперимента, оказывает существенное влияние на стиль программирования, эффективность и надежность программ. Программы, созданные по хорошо продуманным и спроектированным спецификациям гораздо легче отлаживать и тестировать. Поэтому разработчикам больших программ можно порекомендовать книги, посвященные этим вопросам [7, 8, 9]. Опыт преподавания численных методов показывает, что большие трудности у студентов вызывает этап создания программы, реализующей тот или иной метод.

Поэтому дадим несколько советов по конструированию программ. В любой области знаний существует набор методических приемов и правил, которыми должен владеть хороший специалист. Как правило, студент стремится как можно быстрей написать и запустить на выполнение программу, совершенно не заботясь о ее эффективности, читаемости, устойчивости.

Существует программистская мудрость. Каждая программа, (в отличии от программиста), всегда точно знает, что она "должна" делать.

Но, очень часто, создатель программы хотел от нее совсем другого. Хорошим программистом становятся тогда, когда:

- Есть ясное понимание того, что должна делать проектируемая программа.

- Программист может написать такую программу.

Как достичь этого?

- Необходимо соблюдать хороший стиль программирования, т.е. программы должны быть простыми, логичными и легко читаемыми. Программа, которая хорошо структурирована и легко читается, создает впечатление, что автор хорошо знал, что делал. Есть неписанное правило, любой модуль программы должен помещаться на одном листе распечатки.
- Программа должна передавать логику и структуру алгоритма, насколько это возможно.
- Программы с пояснительными комментариями гораздо легче отлаживать.
- Программа должна быть простой, насколько это возможно. Есть принцип KISS (Keep It Simple, Stupid) - делай это проще, дурачок.
- Необходимо выбирать хороший алгоритм решения.
- Необходимо выбирать подходящий язык программирования.

Постулат о необходимости выбора хорошего алгоритма можно проиллюстрировать классическим примером – задачей определения простоты числа N.

Простое число, это такое число, которое делится только на единицу и на себя. Например 1,2,3,7,11 – простые числа.

Первый алгоритм, который приходит в голову – это делить число N на $2, 3, 4, \ldots, N-1$ до тех пор, пока либо N разделится без остатка, либо пока мы не дойдем до N. Очевидно, что это не самый эффективный алгоритм.

Второй алгоритм. Очевидно, что нет необходимости проверять, являются ли все четные числа меньшие N его делителями. Достаточно проверить делимость на два. Дальше проверяем делимость на нечетные числа. Эта идея позволяет сократить объем вычислений почти вдвое. Большинство студентов после этого уже приступает к программированию.

Третий алгоритм. Проведем дальнейший анализ алгоритма. Можно заметить, что достаточно проверить, являются ли делителями числа меньшие или равные $\sqrt{N}$. Если существует делитель, больший квадратного корня из N, то должен существовать делитель меньший $\sqrt{N}$. Таким образом, необходимо проверить являются ли делителями числа 2 и нечетные числа, меньшие $\sqrt{N}$.

Если сравнить количество чисел m, которые мы должны проверить для $N = 1000$, получится, что для алгоритма 1 – m=998, для алгоритма 2 – m=500, для алгоритма 3 – m=16.

1.5. Выбор языка программирования

Правильный выбор языка программирования позволяет либо облегчить процесс создания программ для решения задач численного моделирования, либо серьезно затруднить и замедлить этот процесс.

Наилучшим выходом из этого положения было бы вариант, когда читатель владел бы несколькими языками программирования, тогда он бы сам оценил преимущества либо недостатки того или иного языка, применяемого для численных методов. С нашей точки зрения, наиболее подходящим для создания программ, реализующих задачи численного моделирования является язык Fortran 90. Это подтверждает и анализ литературы. Так, начиная с одной из первых профессиональных книг по Fortran – книги Мак-Кракена [2] существует тенденция связывать численные методы и язык Fortran. Современная версия языка Fortran 90 и его реализация Compaq Visual Fortran (CVF) для Microsoft Visual Studio, или Intel Fortran 9.0 для Microsoft Visual Studio Net или Fortran 90 Sun Studio 12 позволяет использовать встроенный аппарат работы с векторами и матрицами, сечениями и вырезками массивов, динамическими массивами и т.д. Появилась основанная на матричных операциях версия языка HPF - High Performance Fortran – позволяющая использовать его для создания параллельных алгоритмов. Но и сам язык CVF 90, Intel Fortran и Fortran Sun Studio 12 базируется на идеях параллельной обработки данных на процессорах типа SIMD (Single Instuction Miltiplied Data) и выступает как язык параллельного программирования.

Современные вычислительные системы предназначены для решения задач математического моделирования и состоят, как правило, из нескольких десятков, а иногда и нескольких сотен процессоров. Студенты, аспиранты, научные работники уже сейчас должны обладать техникой разработки и создания параллельных алгоритмов и программ. Поэтому одной из педагогических целей данной монографии было стремление показать практическую пользу и выгоду параллельного мышления, даже если сегодня работа осуществляется на однопроцессорной ЭВМ. Научится разрабатывать программы для параллельной обработки информации, используя Fortran 90 и Fortran 95, не так уж и сложно.

Эта книга ориентирована, в частности, и на то, чтобы читатель мог писать хорошие параллельные программы, которые будут работать на компьютерах различной архитектуры и при этом разрабатывать их на однопроцессорной ЭВМ. Особенно актуальна задача разработки параллельных алгоритмов сегодня, в связи с массовым появлением доступных многоядерных процессоров.

Основной моделью параллельных вычислений в Fortran 90 и 95 является параллелизм данных. Это означает, что целый массив данных может быть обработан одной командой. И если у нас есть многопроцессорная ЭВМ, то компилятор сам распределит данные по процессорам.

Пример 1.1 Пусть нам нужно сложить два массива B и C. Фрагмент программы на обычном Fortran' e:

```
program one
  integer,parameter:: n=100, m=100
  integer:: i,j
  real, dimension(n,m) a,b,c
   do i=1,n
    do j =1,m
      a(i,j)=b(i,j)+c(i,j)
    enddo
   enddo
    write(*,*) a
   end
```

На Fortran 95, эту программу можно записать, используя встроенные матричные операторы:

```
program two
  integer,parameter:: n=100, m=100
  real, dimension(n,m) a,b,c
  a = b + c
  write(*,*) a
end
```

Вся идеология программирования, начиная с ЭВМ типа фон Неймана заключалась в построении последовательных алгоритмов. Однако, с появлением параллельных компьютеров, появилась необходимость в параллельном мышлении для разработки алгоритмов и программ, предназначенных для работы на многопроцессорных ЭВМ.

1.5.1. Тестирование кода и его надежность

Все примеры и программы, приведенные в книге проверялись и тестировались для подтверждения их корректности. Несмотря на это мы не даем никаких гарантий того, что программы лишены ошибок и недостатков. Нет уверенности, что программы корректно работают в различных стандартах и для компиляторов различных фирм. На эти программы не следует полагаться, если существует вероятность, что некорректный способ получения результатов может привести к материальному ущербу.

1.5.2. Преимущество Фортрана при создании вычислительных программ

В учебные планы многих университетов, особенно физико – математических, химических и других входит изучение языков программирования. В качестве такого языка выбирается, как правило Паскаль. Это в какой-то мере оправдано. Паскаль приобрел большую популярность, как язык начального обучения программированию. Кроме того, есть определенные области, где Паскаль очень хорошо подходит для формализации определенных задач. Но, к сожалению он не предназначен для серьезных программных разработок в области численных методов. В Паскале отсутствует операции возведения в степень, нет комплексных переменных, ограничен набор составных инструкций, ограниченный набор операторов для работы с массивами. Да он и не предназначен для создания программ в области математического моделирования [13].

Язык программирования Бэйсик не может служить сколько – нибудь серьезной альтернативой для создания программ в области численных методов, из - за относительной примитивности конструкций, очень медленного исполнения даже откомпилированных программ и т.д.

Язык программирования C++ является мощным и полезным языком для создания эффективных программ. Но только в руках опытного программиста. Его отличает меньшая понятность и читаемость программ, что снижает производительность программиста, увеличивает время на разработку программ и вероятность создания ошибок.

С нашей точки зрения, наиболее оптимальным решением в ряде случаев, является написание программ на смеси языков: Visual Fortran и Visual C++. В данной книге даются ссылки на книги по Фортрану.

Глава 2

Ошибки вычислений

2.1. Введение

При проведении численных расчетов на ЭВМ возникает целый ряд ошибок. Ошибки можно разделить на три вида:

1. Ошибки, связанные с представлением целых и вещественных чисел в ЭВМ.

2. Неудачный выбор численного алгоритма. При неправильном выборе небольшие ошибки, возникающие в вычислительном процессе и которыми можно пренебречь при проведении кратковременных расчетов, в длительных расчетах могут привести к разрушительным последствиям.

3. Ошибки в научных вычислениях. Такие ошибки связаны, в основном, с игнорированием существенных особенностей решаемой задачи.

2.2. Ошибки вычислений

При проведении численных расчетов существуют абсолютные и относительные ошибки:

Определение 2.1 *Предположим, что* $\widehat{q}$ *является приближением* q*. Тогда* **абсолютная ошибка** *равна* $E_a = |\widehat{q} - q|$*, а* **относительная ошибка** *равна* $E_o = \frac{|\widehat{q}-q|}{|q|}$*, при условии, что* $q \neq 0$*.*

Пример 2.1 Найти абсолютную и относительные ошибки. Пусть $q = 1000000$, $\widehat{q} = 999996$. Тогда абсолютная ошибка равна $E_a = \|\widehat{q} -$

$q\| = |1000000 - 999996| = 4$, относительная ошибка равна: $E_o = \frac{|\widehat{q}-q|}{|q|} = \frac{4}{1000000} = 0.000004$.

При численном решении научных задач ошибка часто зависит от некоторого параметра h. Обычно это шаг разностной сетки, или шаг, возникающий при разложении в ряд Тейлора. Академик Л.Д. Ландау предложил ввести следующее обозначение $O(h)$ (говорят О большое от h).

Определение 2.2 *Говорят, что функция $f(h)$ имеет порядок сходимости $O(h)$, и обозначается $f(h) = O(g(h))$, если существуют такие константы C и c, что*

$$|f(h)| \leq C|g(h)|, \quad \textit{если} \quad h \leq c.$$

Пример 2.2 Рассмотрим функции $f(x) = x^2 + 1$ и $g(x) = x^3$. Так как $x^2 \leq x^3$ и $1 \leq x^3$ для $x \geq 1$, отсюда следует, что $x^2 + 1 \leq 2x^3$ для $x \geq 1$. Поэтому $f(x) = O(g(x))$.

Теорема 2.1 (Тейлора) *Предположим, что $f \in C^{n+1}[a, b]$. Если x_0 и $x = x_0 + h \in [a, b]$, то*

$$f(x + h) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} h^k + O(h^{n+1}).$$

Пример 2.3 Запишем разностное соотношение для производной функции $f(x)$ в точке x:

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{2.1}$$

Здесь h параметр. Если h мало, то можно приблизительно записать следующее соотношение:

$$f'(x) \cong \frac{f(x + h) - f(x)}{h} \equiv \triangle f(x)$$

$$\triangle f(x) = \frac{f(x + h) - f(x)}{h} \approx \frac{f(x) + hf'(x) + h^2 f''(\xi)/2 - f(x)}{h} = f'(x) + hf''(\xi)/2.$$

Здесь $x < \xi < x + h$.

При этом возникает неустранимая ошибка $hf''(\xi)/2$, вызванная отбрасыванием членов ряда Тейлора. Ее можно записать:

$$f'(x) = \frac{f(x + h) - f(x)}{h} + O(h).$$

Пример 2.4 Рассмотрим представление e^{x^2} в виде ряда Тейлора:

$$e^{x^2} = 1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \ldots + \frac{x^{2n}}{n!} + \ldots \tag{2.2}$$

Вычислим, с двойной точностью, интеграл (2.2). Его значение равно:

$$I = \int_0^{1/2} e^{x^2} dx \approx 0.544987104184. \tag{2.3}$$

Возьмем в (2.2) только пять членов. Найдем точность вычисления определенного интеграла, с помощью разложения в ряд, для функции (2.2):

$$f(x) = 1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \ldots .$$

$$I = \int_0^{1/2}(1 + x^2 + \tfrac{x^4}{2!} + \tfrac{x^6}{3!} + \tfrac{x^8}{4!})dx =$$
$$(x + \tfrac{x^3}{3} + \tfrac{x^5}{5\cdot 2!} + \tfrac{x^7}{7\cdot 3!} + \tfrac{x^9}{9\cdot 4!})\Big|_0^{1/2} = 0.544986720817.$$

Мы нашли приближенное значение интеграла I с точностью до пяти значащих цифр. Остальные цифры приближенного значения интеграла отличаются от (2.3).

2.2.1. Распространение ошибки

Рассмотрим, как могут распространяться ошибки в численных расчетах. Рассмотрим сложение двух целых чисел p и q. Каждое из этих чисел, из-за неточного представления в компьютере, записывается в виде:

$$p + q = (\widehat{p} + \varepsilon_p) + (\widehat{q} + \varepsilon_q) = (\widehat{p} + \widehat{q}) + (\varepsilon_p + \varepsilon_q).$$

Таким образом, для операций сложения, ошибка в сумме двух слагаемых равна сумме их ошибок. Для операции умножения выяснить как распространяется ошибка сложнее. Относительная ошибка произведения $p \cdot q$ приближенно равна сумме относительных ошибок.

Упражнения. 1. Найдите абсолютную и относительную ошибки в следующих примерах.

1. $x = 2.71828182,\ \widehat{x} = 2.7182.$
2. $y = 98350,\ \widehat{y} = 98000.$

3. $z = 0.000068,\ \widehat{z} = 0.00006.$

2. Методом почленного интегрирования вычислите определенный интеграл:

$$\int_0^{1/4} e^{x^2} dx \approx \int_0^{1/4} (1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!}) dx.$$

Сравните с "точным" значением $I \approx 0.2553074606$.

2.3. Представление целых чисел

При проведении расчетов на ЭВМ, используются целые и вещественные числа. Внутреннее представление чисел на компьютере – двоичное, вычисления обычно проводятся в десятичной системе.

В компьютере целые цисла представляются конечным количеством разрядов. Представление числа зависит и от языка программирования. Так, в Фортране 90 целые числа бывают INTEGER(1), INTEGER(2) и INTEGER(4). В случае однобайтового целого числа дипазон чисел на компьютере представлен в диапазоне -128 до +127.

Один из главных факторов. который оказывает значительное влияние на проведение расчетов – это конечное количество целых чисел. Для однобайтовых чисел их всего 256. Вне диапазона -128 +127 чисел, в однобайтовом представлении, не существует. При работе внутри диапазона вычисления проводятся правильно. Если же результаты работы над целыми числами выходят за пределы диапазона, то последствия на разных типах компьютеров и компиляторов будут различными. Одни компиляторы выдают сообщения об ошибке. Другие оставляют числа внутри диапазона. Например, если на Visual Fortran 90 мы от k=-128 отнимем 1, то в результате получим k=127. Такие же результаты мы получим и в других диапазонах целых чисел. Таким образом, при работе с целыми числами около границ диапазона нужно быть предельно внимательными.

Еще одним источником ошибок является операция деления целых чисел. Деление целых чисел дает в результате целое число. То есть в качестве результата операции остаток от деления отбрасывается. Например запись на Фортране:

```
integer  i
i=1/2
```

дает в результате 0, а $i = 11/2 = 5$. Такие ошибки очень часто встречаются у начинающих программистов.

Пример 2.5 Вычислить сумму степенного ряда с точностью 10^{-6} :

$$S(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Пусть $x > 0$. Суммировать ряд нужно до тех пор, пока очередной отброшенный член не станет $|x_k| < \varepsilon$. Для уменьшения количества промежуточных вычислений, найдем отношение двух последовательных членов ряда:

$$\frac{a_k}{a_{k-1}} = \frac{(-1)^k x^{2k+1}(2k-1)!}{(-1)^{k-1}(2k+1)! \cdot x^{2k-1}} = -\frac{x^2}{2k(2k+1)}$$

В данном случае можно составить реккурентное соотношение:

$$a_k = a_{k-1} \frac{(-x^2)}{2k(2k+1)}$$

Программа 2.1 Программа вычисления суммы ряда

```
program series
  implicit none
  integer(4)::k=2
  real(4)::a,s,x=0.5,y
  y=x**2
  a=x
  s=a
 do while( abs(a)>1e-06)
   a=a*y/(k*(k+1))
   s=s+a
   k=k+2
 enddo
  write(*,*) s
 end
```

В результате работы программы получим ответ: $s = 0.5210953$.

2.4. Представление вещественных чисел

Десятичное число с плавающей точкой представлено в компьютере в виде числа с мантиссой a и показателем степени b: $a * 10^b$. Как правило b – целое число. У чисел a и b количество разрядов конечно. Так в Фортране

90 вещественные числа с одинарной точностью REAL(4) представлены в диапазоне $|1.18 \cdot 10^{-38} \ldots 3.4 \cdot 10^{38}|$. Распределение двоичных разрядов между порядком и мантиссой в случае одинарной точности составляет 8 и 24, а для вещественных чисел двойной точности – 11 и 53. Современные процессоры имеют внутренние регистры, состоящие из большего числа разрядов. При необходимости можно использовать эти дополнительные разряды для повышения точности расчетов.

Общее количество вещественных чисел, которые представлены в компьютере **конечно**, в отличии от их количества в математике. Кроме того, это конечное число вещественных чисел распределено неравномерно, в диапазоне от минимального до максимального числа.Так, между каждыми соседними степенями числа 2 находится $\sim 10^{22}$ чисел с плавающей точкой. В диапазоне 2^{300} – 2^{301} их будет $\sim 10^{22}$ и столько же чисел будет между 2^{-300} – 2^{-301} (для чисел с двойной точностью). Таким образом, числа с плавающей точкой гуще расположены вблизи нуля.

Результаты операций над вещественными числами в ЭВМ практически никогда не бывают точными. Это связано с тем, что при проведении арифметических операций с числами постоянно происходят операции округления. Кроме того, при работе с числами, которые отличаются на несколько порядков, происходят операции усечения, то есть отбрасываются последние значащие числа результата.

Определение 2.3 *Разница между точным результатом и решением, полученном на компьютере, называется ошибкой округления.*

Пример 2.6 Рассмотрим сумму:

$$x + y = x(1 + \frac{y}{x}),$$

если $\frac{y}{x} < \varepsilon_m$, то $x + y = x$. Здесь ε_m – машинный *эпсилон* – минимально возможное число, отличное от нуля, которое определено в компьютере.

Операция усечения происходит при проведении расчетов, независимо от того, происходило ли на самом деле округление или просто отбрасывание последних цифр.

Язык Фортран поддерживает спецификацию IEEE Floating Point Standard. Этот стандарт точно определяет, каким образом осуществляется округление, какие инструкции должен выполнить процессор при нестандартных операциях – переполнения, исчезновения порядка и т.д. Для чисел с обычной точностью REAL(4), удовлетворяющих стандарту IEEE, $\varepsilon_m = 2^{-22} \approx 1.2 \cdot 10^{-7}$. Поэтому, при проведении численных расчетов, получить результат с точностью более семи верных десятичных знаков невозможно.

Еще одной возможной ошибкой при проведении численных расчетов, является неточное представление вещественных чисел. Десятичное число 1.0/6.0 не имеет точного двоичного представления в виде числа с плавающей точкой. Кроме того, если мы будем рассматривать оператор присваивания $a = 1/6$, то в ячейке a будет находится 0, так как справа от знака равенства мы записали операцию деления двух целых чисел. В этом случай сначала произойдет целочисленное деление, в результате которого будет отброшена дробная часть и результат, то есть 0, будет записан в ячейку a.

Другая распространенная ошибка, связанная с компьютерным представлением вещественных чисел, заключается в следующем. Рассмотрим следующий пример:

Пример 2.7 В некоторых физических и биологических моделях, при изучении, например, популяции животных используется уравнение вида:

$$p(t) = a \cdot e^{bt},$$

здесь t – время, $p(t)$ – количество особей к моменту времени t, a и b – константы, задающие начальный размер популяции и скорость ее роста соответственно. Допустим, мы хотим узнать, к какому времени популяция достигнет некоторого числа, например c = 1000 особей. Для решения этой задачи нам необходимо решить нелинейное уравнение

$$c - a \cdot e^{bt} = 0.$$

Решения нелинейных уравнений мы будем рассматривать позже (в **Главе 4**). В данном случае, нас будут интересовать только условия выхода из программы. Обычно пишут условие

```
if( p == c) then
  write(*,*) 't = ', t
  stop
end if
```

– завершить расчет и выдать на печать значение t. Это условие, из - за неточного представления вещественных чисел на компьютере, никогда не выполнится. Правильно будет применить другое условие выхода:

```
  if( p >= c )  then
    write(*,*) 't = ', t
    stop
  end if
```

Использование констант при работе с Fortran. В версии Fortran 90 определено большое количество часто используемых констант, которые находятся в библиотеке IMSL. Для их использования необходимо подключить эту библиотеку с помощью команды

```
use imsl
```

В библиотеке IMSL находятся такие константы, как число π, атомная единица массы, постоянная Больцмана, число Авогадро и т.д. Там же находится и машинный эпсилон ε_m. Поэтому, при проведении расчетов, особенно с большой точностью не совсем логично использовать приблизительные значения констант: $\pi = 3.1415$, в этом случае можно верить только четырем цифрам после запятой и решение не даст требуемую точность ε_m.

Упражнения

1. Найти с точностью до 5 значащих цифр:

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^k}{k!}$$

.

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

2. Ряд Тейлора для функции ошибки имеет вид:

$$erf(x) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k x^{2n+1}}{n!(2n+1)}$$

   Этот ряд сходится для всех x. Найти, с точностью до ε_m сумму ряда для x = 0.5, 1.0, 5.0 и 10.0.

3. Вычислите сумму ряда

$$\phi(x) = \sum_{k=0}^{\infty} \frac{1}{k(k+x)},$$

   для x= 0.1,..,1.0.

2.5. Ошибки в научных вычислениях

Кроме ошибок, перечисленных выше, в научных расчетах встречаются и другие ошибки.

1. Ошибки эксперимента. Экспериментальные данные получают с помощью измерительных приборов. Это означает, что при экспериментах всегда присутствует ошибка измерения.

2. Игнорирование особенностей задачи. Существуют плохо обусловленные задачи, которые очень чувствительны к небольшому изменению начальных данных.

Рассмотрим некоторые примеры. Если измерительный прибор дает только две верные цифры, то невозможно получить результат с четырьмя верными значащими цифрами.

Пример 2.8 Рассмотрим вычисление значения e^x с помощью разложения в ряд Тейлора:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

Этот ряд сходится для всех значений $|x| < \infty$. Приведем программу для нахождения суммы этого ряда:

Программа 2.2 Программа вычисления числа e

```
  program exp1
    integer::i=1
    real(4):: x, s=1.0, tend=1.0,eps=1.0E-06
!   Введем значения x
    write(*,*) 'Введите x '
    read(*,*) x
   do while(tend > eps )
     tend = tend*(x/i)
     s = s + tend
     i = i+1
   end do
    write(*,*) 'exp1 = ',s,' exp = ',exp(x)
    end
```

Вычисления заканчиваются в том момент, когда очередной член ряда станет меньше eps. Таблица значений e^x для некоторых значений x выглядит так:

x	сумма	e^x
1	2.718282	2.718282
5	148.4132	148.4132
10	22026.47	22026.46
15	3269017.0	3269017.0
20	$4.8516531 \cdot 10^8$	$4.8516520 \cdot 10^8$
-1	0.3678794	0.3678795
-5	$6.7377836 \cdot 10^{-3}$	$6.7379947 \cdot 10^{-3}$
-10	$-1.6408609 \cdot 10^{-4}$	$4.5399930 \cdot 10^{-5}$

Для значений $x > 0$ эти числа хорошо согласуются с "точным" решением. Но при $x < 0$ результаты значительно хуже.

Пример 2.9 *Вычисление e^x устойчивый алгоритм.* При $x > 0$ предыдущий алгоритм дает приемлимые результаты. Изменим этот алгоритм для отрицательных значений x:

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots}$$

Упражнение. Видоизмените программу для вычисления значения x. Для **Примера 2.9** должны получится следующие результаты:

x	сумма	e^x
1	2.718282	2.718282
5	148.4132	148.4132
10	22026.47	22026.46
15	3269017.0	3269017.0
20	$4.8516531 \cdot 10^8$	$4.8516520 \cdot 10^8$
-1	0.3678794	0.3678795
-5	$6.7379461 \cdot 10^{-3}$	$6.7379947 \cdot 10^{-3}$
-10	$4.5399924 \cdot 10^{-5}$	$4.5399930 \cdot 10^{-5}$

Теперь результаты выглядят значительно лучше. Данный пример показывает, что выбор алгоритма численного расчета имеет большое значение.

Другой причиной возникновения ошибок при проведении научных расчетов служат плохо обусловленные задачи. Приведем пример такой задачи [1].

Пример 2.10 Пример плохо обусловленной задачи. Найдем все корни полинома четвертой степени:

$$x^4 - 4x^3 + 8x^2 - 16x + 15.999999999 = (x-2)^4 - 10^{-8} = 0. \qquad (2.4)$$

Корни данного уравнения удовлетворяют уравнению:

$$(x-2)^2 = \pm 10^{-4}, \text{ отсюда следует:}$$

$$x - 2 = \pm 10^{-2} \text{ и}$$

$$x - 2 = \pm 10^{-2} i.$$

Отсюда следует, что $x_1 = 2.01,\ x_2 = 1.99,\ x_3 = 2 + 0.01i,\ x_4 = 2 - 0.01i$

Если на компьютере машинный эпсилон $\varepsilon_m > 10^{-8}$, то свободный член будет округлен до 16.0 и, вместо уравнения (2.4) будет решаться уравнений $(x-2)^4 = 0$. У этого уравнения все четыре корня равны 2, что отличается от корней уравнения (2.4).

В данном примере мы видим, что малое изменение одного коэффициента (на 10^{-8}) приводит к значительному изменению решения, причем это происходит независимо от метода решения.

Пример 2.11 Ошибки округления для конечных разностей. Рассмотрим **Пример 2.3** и проанализируем, как влияют ошибки округления на вычисление производной $f'(x)$.

Будем полагать, что x и x+h не содержат ошибок округления. Тогда ошибка в разностном отношении $\triangle_k f(x)$ будет ограничена величиной $2|f(x)|\varepsilon_m/h$. Отсюда, и из (2.2) следует, что неустранимая ошибка + ошибки округления, то есть полная ошибка есть:

$$|err| \le E_{full} = \frac{h}{2}|f''(\xi)| + \frac{2}{h}|f(x)|\varepsilon_m = O(h) + O(1/h). \tag{2.5}$$

Минимальное значение шага h можно получить, если продифференцировать E_{full} по h и приравнять производную нулю, тогда:

$$h = 2\sqrt{\frac{|f(x)|\varepsilon_m}{|f''(\xi)|}} \tag{2.6}$$

Это значение шага дает оценку (2.5). Из (2.5) видно, что ошибка, с уменьшением h сначала убывает, а затем, начиная с некоторого значения h начинает возрастать.

Более точное приближение для производной можно получить, если уменьшить либо ошибку округления, либо неустранимую ошибку. Ошибку округления можно уменьшить, используя повышенную точность. Для компьютеров на базе x86 и Фортран90 можно перейти на двойную точность. Современные процессоры позволяют использовать то обстоятельство, что их внутренние регистры имеют 80 разрядов и больше, и этим

можно воспользоваться программным путем. При этом уменьшится значение ε_m, в связи с чем уменьшится ошибка округления, а следовательно и E_{full}.

Другой путь, это уменьшение неустранимой ошибки без уменьшения шага h. Для этого, при проведении вычислений, можно воспользоваться не левой разностью, как в (2.1), а центральной разностью:

$$\delta_n f(x) = \frac{f(x+h) - f(x-h)}{2h},$$

если проделать те же выкладки, что и в (2.2) то мы получим:

$$\delta_n f(x) = f'(x) + \frac{h^2}{3!} f'''(x) + \frac{h^4}{5!} + \dots$$

В этом случае неустранимая ошибка будет порядка $O(h^2)$.

Ошибки при проведении численных расчетов присутствуют всегда! Основная задача заключается в минимизации этих ошибок. Для хорошо обусловленной задачи плохой алгоритм (**Пример 2.5**) может дать неприемлимые результаты. С другой стороны, **Пример 2.10** показывает, что существуют задачи, для которых невозможно подобрать хороший алгоритм, так как они чуствительны к очень маленьким ошибкам во входных данных и к накоплению ошибок округления. При проведении аналитических расчетов ошибок округления нет. Таким образом, можно сделать вывод, что при проведении научных расчетов возникают ошибки, связанные с неустойчивостью алгоритмов и с плохо обусловленными задачами. Имея представление об этих источниках ошибок, можно легче конструировать устойчивые алгоритмы решения задач.

2.6. Отладка и тестирование программ

Проблемам отладки и тестирования программ не всегда уделяется должное внимание. Мы сознательно включили этот параграф в главу об ошибках вычислений. Традиционно, начинающие программисты, на отладку и тестирование программы выделяется очень мало времени или не выделяется совсем. Хотя любой опытный разработчик программ знает, что на отладку и тестирования сколь – нибудь серьезных программ необходимо отвести не менее половины всего времени, планируемого на создание программы. Чем с большим вниманием отнестись к задаче отладки и тестирования, тем меньше неприятных сюрпризов встретится при создании

программы. Разберемся сначала с терминологией. Отладка и тестирование это две разные фазы поиска и устранения ошибок.

Определение 2.3 *Отладкой называется процесс поиска и устранения синтаксических ошибок и ошибок этапа линкования (или этапа редактирования).*

После завершения этапа отладки программа начинает выдавать какие - то результаты. Неопытные программисты, испытав радость по поводу получения результата, на этом обычно и успокаиваются. На самом деле самое трудное только начинается.

Определение 2.4 *Тестированием называется процесс устранения семантических и логических ошибок и получение результатов расчетов, совпадающих с* **предварительно подготовленными** *аналитическими решениями или экспериментальными данными.*

Остановимся более подробно на этих этапах. (Более подробную информацию можно получить в книгах [8, 9].

Первая стадия – *отладка синтаксиса.* При компиляции проекта в Фортран 90, по умолчанию, создается отладочная (Debug) конфигурация проекта. В отладочной конфигурации содержится полная информация, необходимая для символьной отладки в терминах языка Фортран 90. При этом пользователю представляется возможность работать в отладочном режиме как в объектном коде, так и с операторами языка Фортран 90. Первый этап это устранение синтаксических ошибок и предупреждений (Warning), выдаваемых компилятором. Компилятор, в окне Output выдает информацию об имени компилируемого файла, ошибочную строку, порядковый номер ошибки и краткое описание этой ошибки. Такая же информация выдается и для предупреждений. Количество ошибок, выдаваемых компилятором за одну компиляцию ограничено 30 ошибками (это значение можно изменить). Как правило, часть ошибок бывают "наведенными", то есть они появляются вследствии других ошибок, сделанных в предыдущих операторах. Поэтому исправление ошибок следует начинать с первой ошибки и двигаться по списку ошибок последовательно, сверху вниз.

Ошибки линкования. (Синонимы: редактирования, связывания). После устранения синтаксических ошибок, наступает пора ошибок линкования. Редактор связей (linker) выдает информацию об имени программы, номер ошибки и ее краткое описание (полное описание можно посмотреть в Help'е). Типичные ошибки этого этапа – это отсутствие подпрограммы, на которую есть ссылка, либо отстутствие объявления массива. В последнем случае обращение к имени массива воспринимается, как обращение

к подпрограмме, которой, естественно, нет в данном проекте. Очень часто причиной возникновения ошибок линкования является отсутствие подключения необходимых библиотек, например Array Visualiser, IMSL, OpenGL и так далее.

Тестирование. Сразу после завершения этапа отладки, программа очень редко начинает выдавать правильные результаты. И, как правило, это результат гораздо более трудноустранимых ошибок. Как это не парадоксально звучит, если сразу после отладки программа начинает выдавать правдоподобные результаты – это не повод для радости. Как показывает практика в таких случаях процесс тестирования может затянуться надолго.

В ходе разработки проекта программы необходимо разработать и систему тестов. Все эти тесты должны быть направление на выявление, по возможности, всех ошибок, а не доказательство того, что со специально (или случайно) подобранными исходнымии начальными данными программа дает правильный результат. Иными словами, система тестов должна помочь выявить ошибки в логике работы программы и помочь их устранить. Эту работу нельзя унифицировать и стандартизировать. Но необходимо проверить, как ведет себя программа на границах исследуемого диапазона, чтобы убедится, что не происходит выхода за пределы массивов. Нужно также проверить, как программа работает с отрицательными значениями (в качестве примера – задача с неустойчивым алгоритмом нахождения $exp(x)$), при отрицательных значениях x и т.д.

Наличие встроенного в Visual Fortran отладчика позволяет облегчить и ускорить процесс отладки. Опытные программисты, с большим стажем работы, вставляли в текст программы специальные вставки, которые позволяли контролировать процесс выполнения программы. Использование отладчика позволяет существенно упростить (технологически, а не логически) процесс поиска семантических и логических ошибок. Процесс отладки лучше всего разбить на несколько этапов.

1. Установить в теле программы точки прерывания.

2. Задать события, при наступлении которых необходимо остановить выполнение программы.

3. Задать вывод в окно Watch тех величин, которые требуют особого контроля.

После того, как указанные действия выполнены необходимо запустить программу на выполнение в режиме отладки. Это можно сделать

с помощью клавиши F5, или нажатием соответствующей кнопки на инструментальной панели.

Точку останова или контрольную точку можно установить нажатием клавиши F9, либо соответствующей кнопки на инструментальной панели. Слева от выбранного операторы появиться красная точка. Повторное нажатие этой же кнопки убирает точку отладки.

В режиме отладки появляется панель, которая позволяет управлять процессом выполнением программы. Панель имеет два ряда кнопок, которые задают следующие режимы, верхний ряд:

1. Перезапуск программы.
2. Завершить отладку.
3. Прервать выполнение.
4. Изменить код программы.
5. Перейти к следующему оператору.
6. Войти в блок.
7. Обойти блок.
8. Выйти из блока.
9. Выполнить до курсора.

В нижнем ряду задаются следующие режимы:

1. Быстрая оценка.
2. Контроль.
3. Переменные.
4. Регистры.
5. Память.
6. Стек вызовов.
7. Ассемблерный код.

Кнопки 1 и 2 позволяют осуществить повторный старт программы и остановку режима отладки. Кнопка 3 позволяет прервать выполение программы, в частности, если произшло зацикливание. Кнопка 4 дает возможность внести изменения в код программы. Управление ходом выполнения программы осуществляется кнопками 5 – 9. Кнопка 5 позволяет перейти к следующему оператору, кнопка 6 – выполнить следующий блок с заходом в блок то есть в подпрограмму или функцию. Кнопка 7 позволяет выполнить блок без захода в него. Кнопка 8 осуществляет выход их блока. Кнопка 9 позволяет осуществить выполнение всех операторов, от текущего до того оператора, на который мы установили курсор. Кнопка 1 во втором ряду позволяет в появившейся панели задать арифметическое выражение со значениями, посчитанными к данному моменту и оценить результат.

Внизу рабочей области расположены окна Variables и Watch. В окне Variables отображаются значения локальных переменных для той программной единицы, с которой мы работаем в текущий момент. Перед именем массива расположена кнопка +, которая позволяет развернуть массив и показать содержимое его ячеек. В окне Watch отображаются значения переменных или выражений. Имена этих переменные можно набрать непосредственно в поле Name или просто "перетащить" мышкой из главного окна. Окно Watch содержит внизу четыре закладки Watch1 – Watch4. В любом из этих окон можно составить свой список переменных и вызывать его по мере работы программы.

2.7. Упражнения

1. Написать и отладить программу, расчитывающую с точностью ε_m сумму ряда:

$$I_n(x) = \frac{x^n}{2^n n!} - \frac{x^{n+2}}{2^{n+2}(n+1)!} + \frac{x^{n+4}}{2^{n+4}2(n+2)!} - \frac{x^{n+6}}{2^{n+6}6(n+3)!} + \dots$$

   Исследуйте влияние ошибок округления.

2. Написать и отладить программу, которая подсчитывает сумму ряда. Указание: Подумайте о скорости сходимости этого ряда.

$$q(x) = \sum_{k=1}^{\infty} \frac{1}{k(k+x)}, \quad x = 0 \div 1.0.$$

   Расчет произвести с точностью $\varepsilon \le 0.5 \cdot 10^{-6}$.

3. Найдите сумму ряда с точностью $\varepsilon \leq 0.5 \cdot 10^{-6}$

$$q(x) = \sum_{k=1}^{\infty} \frac{1}{4k^2 - 1}$$

4. Формула приближенного вычисления числа π имеет вид:

$$P_{k+1} = 2^k \sqrt{1(1 - \sqrt{1 - (p_k/2^k)^2})}, \quad P_2 = 2\sqrt{2}.$$

   Вычислите значения π для $n \leq 40$

5. Найдите полином Тейлора степени $n = 4$ в окрестности точки x_0 для следующих функций:

   - $f(x) = \sqrt{x}, \quad x_0 = 1.$
   - $f(x) = cos(x), \quad x_0 = 0.$
   - $f(x) = x^5 + 4x^2 + 3x + 1, \quad x_0 = 0.$

6. Вычислить бесконечное произведение с точностью $\varepsilon \leq 0.5 \cdot 10^{-6}$.

$$B(x) = \frac{1}{x} \prod_{k=1}^{\infty} \frac{(1 + 1/k)^2}{1 + x/k}.$$

7. Найдите эквивалентную формулу, которая позволяет избежать потери точности при вычислении.

   - $cos^2(x) - sin^2(x),$ для $x \approx \pi/4.$
   - $\sqrt{x^2 + 1},$ для больших x.
   - $ln(x + 1) - ln(x),$ для больших x.

Глава 3

Решение систем линейных уравнений

3.1. Введение

Приведем основные сведения из курса линейной алгебры, которые нам понадобятся в дальнейшем.

Определение 3.1 *Прямоугольной матрицей порядка* $m \times n$ *называется прямоугольная таблица в виде:*

$$A = a_{ij} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Определение 3.2 *Единичной называется матрица* e_{ij}, *у которой*

$$e_{ij} = \begin{cases} 0, \text{если } i \neq j, \\ 1, \text{если } i = j. \end{cases}$$

$$E = e_{ij} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Сумма двух матриц размера $m \times n$ есть матрица размера $m \times n$:

$$A + B = \sum_{i,j=1}^{n,m} (a_{ij} + b_{ij}).$$

Произведение матрицы на число β тоже является матрицей размера $m \times n$:

$$\beta \times A = \beta \times a_{ij}, \text{ где } i = 1, n, \ j = 1, m.$$

Определение 3.3 *Произведение (левое) матрицы A размерности $m \times n$ на матрицу B размерности $n \times k$ является матрицей C размерности $m \times k$.*

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}.$$

Любой элемент матрицы C – c_{ik} является суммой произведения i-ой строки матрицы A на соответствующие элементы k-го столбца матрицы B. В общем случае правого произведения $B \times A$ не существует.

Для квадратных матриц $n \times n$ существует как левое произведение $A \times B$, так и правое произведение $B \times A$, и в общем случае, $A \times B \neq B \times A$.

Для квадратной матрицы вводится понятие определителя.

$$detA = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

$$detA = \sum_{j=1}^{n} a_{ij} A_{ij}, \quad \text{где } A_{ij} \text{ алгебраическое дополнение элемента } a_{ij}.$$

Определение 3.4 *Матрица называется диагональной, или ленточной, если на главной диагонали и сверху и снизу от главной диагонали элементы матрицы $a_{ij} \neq 0$, а все остальные элементы $= 0$.*

Пример 3.1. Примером ленточной матрицы с шириной ленты равной 3 является трехдиагональная матрица:

$$A = a_{ij} = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & a_{nn-1} & a_{nn} \end{bmatrix}$$

Другими словами ленточная матрица, это матрица, все ненулевые элементы которой находятся вблизи главной диагонали: $a_{ij} = 0 \ \forall i, j : |i - j| > m, \ m \ll n$. Ширина ленты $l = 2m + 1$. При m=1 – это трехдиагональная матрица.

Такие ленточные матрицы используются для решения дифференциальных уравнений неявными методами.

Определение 3.5 *Квадратная матрица A называется симметричной относительно главной диагонали, если элементы этой матрицы удовлетворяют выражению* $a_{ij} = a_{ji}, \forall i, j = 1 \div n$.

Определение 3.6 *Транспонированной матрицей называется матрица у которой строки и столбцы поменялись местами:*

$$A^T = a_{ij}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\ a_{12} & a_{22} & a_{32} & \cdots & a_{n2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & a_{3n} & \cdots & a_{nn} \end{bmatrix}$$

Определение 3.7 *Если* $A = A^T$, *то такая матрица называется симметричной.*

Определение 3.8 *Матрица называется верхней треугольной матрицей, если все элементы, лежащие под главной диагональю равны нулю.*

Определение 3.9 *Матрица называется нижней треугольной матрицей, если все элементы, лежащие над главной диагональю равны нулю.*

Определение 3.10 *Матрица* A^{-1} *называется обратной к матрице A, если* $A \cdot A^{-1} = E$. Для существования обратной матрицы существует необходимое и достаточное условий: $detA \neq 0$.

Определение 3.11 *Определитель n-го порядка, который составлен из элементов, находящихся на пересечении m строк и m столбцов матрицы A называется минором.*

Определение 3.12 *Рангом матрицы A называется число r такое, что все миноры порядка* $\geq (r+1)$ *равны нулю.*

Определение 3.13 *Характеристическим уравнением матрицы A называется уравнение вида:*

$$detA \begin{bmatrix} a_{11} - \lambda_1 & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda_2 & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda_n \end{bmatrix} = 0.$$

Корни λ_i называются собственными значениями матрицы A. $det(A - \lambda E)$ называются характеристическим полиномом. Для исследования предлагаемых алгоритмов решения систем линейных уравнений нам потребуются некоторые сведения из аппарата линейных нормированных пространств.

Определение 3.14 *Линейным нормированным пространством называется множество, в котором определены операции сложения элементов, умножения элемента на число и введено понятие нормы для каждого элемента.*

Определение 3.15 *Нормой будем называть вещественное неотрицательное число, которое удовлетворяет следующим условиям:*

1. $\|x\| > 0$, если $x \neq 0$, $\|0\| = 0$, и при $x = 0$, $\|x\| = 0$
2. $\|cx\| = |c| \cdot \|x\|$, $\forall c \in \mathbb{R}$,
3. $\|x + y\| \leq \|x\| + \|y\|$.

В определенном смысле норма обладает многими аналитическими свойствами длины в евклидовом пространстве. Приведем несколько примеров норм:

1. $\|x\|_1 = \sum_{i=1}^{n} |x_i|$
2. $\|x\|_2 = (\sum_{i=1}^{n} |x_i|^2)^{\frac{1}{2}}$
3. $\|x\|_\infty = \max\limits_{\forall i} |x_i|$

Это примеры норм для линейного нормированного пространства, элементами которого являются векторы.

Можно определить матричную 1 норму:

$$\|A\|_1 = \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|.$$

Можно вести другую норму :

$$\|A\| = M = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}.$$

В пространстве C – множества определенных и непрерывных при $0 \leq t \leq 1$ функций $f(t)$, норма $\|x\|_c = max|x(t)|$ называется Чебышёвской нормой. Определим понятие предела последовательности векторов и матриц. Рассмотрим последовательность векторов: $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ с компонентами $x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}$. Если существует предел последовательности:

$$L = \lim_{l \to \infty} (x_i^{(l)}),$$

то вектор с компонентами $x_1, x_2, \ldots, x_n$ называется пределом последовательности $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$. Аналогичным образом можно определить и предел последовательности матриц. Для сходимости последовательности матриц $x^{(k)}$ к x необходимо и достаточно, чтобы выполнялось условие: если $\|x^{(k)} - x\| \to 0$, то и $\|x^{(k)}\|$ сходится к $\|x\|$.

3.2. Метод простой итерации

Одним из простых методов решения систем линейных уравнений являтся метод простой итерации или итерацией Якоби. Возьмем систему линейных уравнений

$$\begin{cases} 4x_1 + x_2 - x_3 & = 13 \\ x_1 - 5x_2 - x_3 & = -8 \\ 2x_1 - x_2 - 6x_3 & = -2. \end{cases}$$

Из первого уравнения можно получить:

$$x_1 = \frac{13 - x_2 + x_3}{4}, \tag{3.1}$$

Из второго уравнения получим:

$$x_2 = \frac{-8 + x_1 - x_3}{5}, \tag{3.2}$$

Из третьего уравнения получим:

$$x_3 = \frac{-2 + 2x_1 - x_2}{6}. \tag{3.3}$$

Идея метода простой итерации состоит в том, чтобы использовать уравнения (3.1) – (3.3) для получения следующих итераций:

$$\begin{aligned} x_1^{(k+1)} &= \frac{13 - x_2^{(k)} + x_3^{(k)}}{4}, \\ x_2^{(k+1)} &= \frac{-8 + x_1^{(k)} - x_3^{(k)}}{5}, \\ x_3^{(k+1)} &= \frac{-2 + 2x_1^{(k)} - x_2^{(k)}}{6}. \end{aligned} \tag{3.4}$$

Для начала итераций необходимо задать начальную точку $\vec{x}^0 = \{0, 1, 1\}$. Можно убедится, что решением системы уравнений (3.1) является вектор $\vec{x} = \{3, 2, 1\}$

Метод простой итерации сходится не всегда. Введем следующие определения.

Определение 3.16 *Матрица A размером $n \times n$ является* ***строго диагонально доминирующей,*** *если:*

$$|a_{kk}| > \sum_{j=1, j\neq k}^{n} |a_{kj}|, \quad k = 1, 2 \ldots, n, \tag{3.5}$$

то есть величина элемента на главной диагонали должна превышать сумму величин всех остальных элементов.

Определение 3.17 *Если A – строго доминирующая матрица, тогда уравнение $Ax = b$ имеет единственное решение.*

3.2.1. Итерации Гаусса – Зейделя

Можно ускорить метод простой итерации, если воспользоваться уже найденными значениями $x_i^{(k+1)}$:

$$\begin{aligned} x_1^{(k+1)} &= \frac{13 - x_2^{(k)} + x_3^{(k)}}{4}, \\ x_2^{(k+1)} &= \frac{-8 + x_1^{(k+1)} - x_3^{(k)}}{5}, \\ x_3^{(k+1)} &= \frac{-2 + 2x_1^{(k+1)} - x_2^{(k+1)}}{6}. \end{aligned} \tag{3.6}$$

Программа 3.1 Программа решения системы линейных уравнений методом простой итерации

```
    program Iteration
      implicit none
! Variables
      integer,parameter::nx=3
      integer::i,j,n
      real::a(nx,nx),b(nx),s(nx),ss,x(nx),eps=1.0E-06
      a=reshape((/ 4., 1.,-1.,  &
                   1.,-5.,-1.,  &
                   2.,-1.,-6. /), shape=(/nx,nx/),order=(/2,1/))
```

```
      b=(/13.,-8.,-2./)
    ! Body of Iteration
      x=(/1,1,2/)
      do
       do i = 1,nx
        do j = 1,3
          ss = ss+(-a(i,j)*x(j))
        enddo
         s(i) = ss+b(i)
         ss = 0.
!        write(*,*) ss
         x(i) = (s(i)+a(i,i)*x(i))/a(i,i)
         n = n+1
       enddo
      if( abs(sum(s-x))<eps ) exit
      if( n>25 ) exit
        s=x
      enddo
      open(7,file='Jacoby.dat')
      write(7,*) n
      write(7,91) (x(i),i=1,nx)
  91  format(3(2x,f11.6))
    end program Iteration
```

Результат работы программы.

```
       Number of iterations   51
        x1            x2           x3
     3.000000     2.000000     1.000000
```

Упражнение: Переделать программу метода простой итерации для метода Гаусса – Зейделя

3.3. Метод Гаусса

Будем искать решение системы линейных уравнений

$$Ax = b, \tag{3.7}$$

здесь $A = a_{ij}$ – квадратная матрица порядка n, и векторы x и b также имеют порядок n. Предположим также, что матрица A невырожденная.

Решение уравнения (3.7) будем искать методом исключения Гаусса. Для успешной работы метода Гаусса необходимо использовать идею выбора главного элемента. В основе метода исключения Гаусса лежит идея приведения системы уравнений (3.7) к системе более простого вида – треугольной. Покажем это на примере:

Пример 3.2. Найти решение системы уравнений:

$$\begin{cases} 2x_1 + 4x_2 + x_3 & = -1 \\ 16x_2 + 11x_3 & = -5 \\ 107x_3 & = 107 \end{cases}$$

В матричной форме оно запишется так:

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 16 & 11 \\ 0 & 0 & 107 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1 \\ -5 \\ 107 \end{pmatrix}$$

Матрица A из (3.7) является верхнетреугольной. Из последнего уравнения находим $x_3 = +1$. Подставив x_3 во второе уравнение (3.8) получим $x_2 = -1$. И, наконец, подставив x_2 и x_3 в (3.8) получим $x_1 = +1$. Проверить правильность полученного решения можно подставив вектор решения $(+1, -1, +1)$ в (3.8).

При решении мы воспользовались тем обстоятельством, что матрица A имеет треугольный вид. Это так называемый обратная подстановка метода Гаусса.

Идея метода исключения Гаусса состоит в том, чтобы на первом этапе, путем невырожденных преобразований, привести исходную матрицу A к верхнетреугольному виду. Решение треугольной матрицы можно получить с помощью уравнений:

$$x_i = \begin{cases} b_n/a_{nn}, & \text{если } i = n, \\ (b - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii}, & \text{если } i \neq n. \end{cases} \tag{3.8}$$

Этот алгоритм требует $O(n^2/2)$ операций. Алгоритм применим к любой системе уравнений, у которой диагональные элементы отличны от нуля.

Если бы мы воспользовались решением системы уравнений по правилу Крамера, потребовалось бы приблизительно $n \cdot n^2$ арифметических операций.

Прямой ход метода Гаусса состоит из n-1 шага. На i-ом шаге кратные i-го уравнения вычитаются из оставшихся уравнений для исключения i-го неизвестного. Всего для прямого хода требуется приблизительно $O(n^3/3)$. Рассмотрим еще один пример.

Пример 3.3. Решить систему уравнений:

$$\begin{cases} 2x_1 + 4x_2 + x_3 & = -1 \\ x_1 - 6x_2 - 5x_3 & = 2 \\ 2x_1 + 3x_2 + 7x_3 & = 6 \end{cases} \tag{3.9}$$

На первом шаге из второго и третьего уравнения исключим, с помощью первого уравнения x_1. Для этого умножим второе уравнение на -2 и сложим с первым уравнением. Третье уравнение вычтем из первого. В результате получим:

$$\begin{aligned} 2x_1 + 4x_2 + x_3 &= -1 \\ 6x_2 + 11x_3 &= -5 \\ x_2 - 6x_3 &= -7 \end{aligned} \tag{3.10}$$

Умножим третье уравнение (3.10) на -16 и сложив со вторым, получим:

$$\begin{aligned} 2x_1 + 4x_2 + x_3 &= -1 \\ 16x_2 + 11x_3 &= -5 \\ 107x_3 &= -107 \end{aligned} \tag{3.11}$$

В результате у нас получилась верхнетреугольная матрица (3.8), решение которой мы уже нашли.

В общем виде метод Гаусса можно определить так:

1. Приводим матрицу системы линейных уравнений к верхнетреугольному виду – прямой ход.

2. Путем подстановки найденных значений, приводим верхнетреугольную матрицу к единичной – обратный ход.

Основные формулы метода Гаусса. Пусть у нас остались члены с ненулевыми элементами, лежащими ниже главной диагонали.

$$\sum_{j=l}^{n} a_{ij}^{(l)} x_j = b_i^{(l)}, \quad l \le i \le n$$

Умножим первую строку на число $d_{ml} = \frac{a_{ml}^{(l)}}{a_{ll}^{(l)}}, \quad m > l$. Затем вычтем эту строку из m строки. После этого, первый ненулевой элемент будет равен нулю (чего мы и добивались). Остальные элементы примут вид:

$$\begin{aligned} a_{mk}^{(l+1)} &= a_{ml}^{(l)} - d_{ml} a_{lk}^{(l)}, \\ b_m^{(l+1)} &= b_m^{(l)} - d_{ml} b_l^{(l)}, \quad l < m. \end{aligned}$$

Обратим в нуль все элементы столбца l, которые лежат ниже главной диагонали, с помощью уравнений (3.12). Уравнения для обратного хода мы уже рассмотрели в (3.8).

В рассмотренном выше примере, при приведении матрицы к верхнетреугольному виду, нам было необходимо провести деление на элемент $a_{ll}^{(l)}$. Такие диагональные элементы верхнетреугольной матрицы называют *главными элементами*. первый главный элемент это коэффициент при первом неизвестном в первом уравнении. В нашем примере главными элементами будут 2, 16 и 107. Если один из главных элементов равен нулю, то алгоритм метода Гаусса становится невозможным. Очевидно, что если один из главных элементов не равен нулю, но близок к нему – можно ожидать, что могут возникнуть большие ошибки округления.

Чтобы избежать этого, среди элементов столбца $a_{lk}^{(l)}$ для всех промежуточных вычислений выбирают наибольший по модулю элемент и меняют его местами с главным элементом. Такой метод называют методом Гаусса с выбором главного элемента.

3.3.1. Плохая обусловленность систем линейных уравнений

Как уже отмечалось выше, равенство определителя системы линейных уравнений нулю приводит либо к отсутствию решения (при $b \neq 0$), либо к неединственности решения (при b=0). Большой интерес вызывают системы, у которых определитель системы близок к нулю – это почти вырожденные системы.

Введем более точную и надежную меру близости к вырожденности, используя понятие нормы. Если умножить матрицу A на вектор x получим новый вектор Ax, норма которого может сильно отличаться от нормы вектора x. Изменение нормы прямо связано с той чуствительностью, которую мы хотим измерить. Пусть

$$M = \max_x \frac{\|Ax\|}{\|x\|}, \quad \text{или} \quad \|Ax\| \leq M\|x\|,$$
$$m = \min_x \frac{\|Ax\|}{\|x\|}, \quad \text{или} \quad \|Ax\| \geq m\|x\|.$$

Здесь максимум и минимум берутся по всем ненулевым элементам вектора x. Если матрица вырождена это означает, что m=0.

Определение 3.18 $\frac{M}{m}$ *называется числом обусловленности матрицы* A

$$Q(A) = \frac{\max\limits_x \frac{\|Ax\|}{\|x\|}}{\min\limits_x \frac{\|Ax\|}{\|x\|}}, \tag{3.12}$$

математически это эквивалентно следующему уравнению:

$$Q(A) = \|A\| \cdot \|A^{-1}\|.$$

Из (3.12) следует, что $M \geq m$, поэтому $Q(A) \geq 1$. Рассмотрим диагональную матрицу:

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}.$$

Для нее, так как

$$Q(D) = \frac{max\|D_{ii}\|}{min\|D_{ii}\|} \Rightarrow M = 5,\ m = 1.$$

Отсюда следует, что число обусловленности матрицы: $Q(D) = 5/1 = 5$. Можно показать, что число обусловленности Q(A) играет роль коэффициента увеличения относительной ошибки. Рассмотрим систему алгебраических уравнений:

$$Ax = b. \tag{3.13}$$

Изменим эту систему на величину $\triangle x$:

$$A(x + \triangle x) = b + \triangle b, \tag{3.14}$$

здесь $\triangle x$ – ошибка в определении x, а $\triangle b$ – ошибка в задании b.

$$A(\triangle x) = \triangle b.$$

Из (3.12) и (3.14) следует, что $\|Ax\| = \|b\| \leq M\|x\|$ и $\|A\triangle x\| = \|\triangle b\| \geq m\|\triangle x\|$. Здесь $\frac{\|\triangle b\|}{\|b\|}$ относительное изменение правой части, а $\frac{\|\triangle x\|}{\|x\|}$ – относительная ошибка. Тем самым мы показали, что число обусловленности есть коэффициент увеличения относительной ошибки. Матрицы с

большими числами обусловленности дают большие ошибки при решении систем. Для практических применений можно принять следующее правило: если $Q(A) = 10^4$, а $\varepsilon_m = 10^{-7}$ то мы можем рассчитывать только на приблизительно три верных знака в решении.

Пример 3.4. Пример плохой обусловленности. Рассмотрим систему уравнений

$$\begin{aligned} x + 2y &= 2 \\ 2x + 3y &= 3.4 \end{aligned}$$

Можно проверить, что подстановка значений $x_0 = 1.00$ и $y_0 = 0.48$ в эту систему уравнений, дает решение очень близкое к реальному:

$$\begin{aligned} 1 + 2 \cdot (0.48) - 2 &= -0.04 \approx 0 \\ 2 \cdot 1 + 3 \cdot (0.48) - 3.4 &= 0.04 \approx 0 \end{aligned}$$

Расхождение с точным решением $x_0 = 0.8$ и $y_0 = 0.6$ всего ± 0.04. Поэтому для плохо обусловленных систем вычисления необходимо выполнять с двойной точностью. Когда необходимо решить системы, состоящие из нескольких десятков уравнений, последствия могут быть еще хуже. Плохую обусловленность трудно обнаружить. Для этого нужно решить систему уравнений с начальными данными и с немного измененными начальными данными. Если решения сильно отличаются – это плохо обусловленная система.

3.3.2. Программа решения системы линейных уравнений методом исключения Гаусса

При разработке программ на Фортране необходимо помнить, что в Фортране матрицы в памяти храняться по столбцам, а не по строкам, поэтому желательно организовывать алгоритмы, в которых во внутренних циклах быстрей всего менялся бы внутренний индекс. Пусть мы хотим умножить матрицу на вектор:

$$b = Ax = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}.$$

Обычно вектор вычисляется следующим образом:

$$b = \begin{pmatrix} 1*10 + 2*11 + 3*12 \\ 4*10 + 5*11 + 6*12 \\ 7*10 + 8*11 + 9*12 \end{pmatrix} = \begin{pmatrix} 68 \\ 167 \\ 266 \end{pmatrix}.$$

В этом алгоритме, записанном в кодах Фортрана:

```
do i=1,n
  b(i)=0.0
 do j=1,n
    b(i)=b(i)+A(i,j)*x(j)
 enddo
 enddo
```

Компьютер обращается к матрице по строкам, то есть не так, как они расположены в памяти. Для ускорения вычислений к матрице A следует обращаться по столбцам, что ускорит выполнение программы, особенно для больших матриц. Вычислим вектор b следующим образом:

$$b = 10\begin{pmatrix}1\\4\\7\end{pmatrix} + 11\begin{pmatrix}2\\5\\8\end{pmatrix} + 12\begin{pmatrix}3\\6\\9\end{pmatrix}.$$

Соответствующая часть кода на Фортране будет выглядеть так:

```
b(i)=0.0
do j=1,n
 do i=1,n
  b(i)=b(i)+A(i,j)*x(j)
 enddo
enddo
```

Как мы видим, изменения в программе минимальны, зато быстродействие увеличилось.

Приведем несколько вариантов программы. Первый вариант самый простой и логически понятный. Второй вариант создан с использованием возможностей языка Фортран 90 [16]. Третий вариант написан с использованием библиотеки IMSL.

Программа 3.2 Метод Гаусса. 1 вариант

```
module sq
 real,allocatable::a(:,:),x(:),b(:)
 real::eps=1.0e-7
 integer::i,j,k,l,m,n
contains
 subroutine  gauss1()
  m=n+1
```

```
  allocate(a(n,m),x(n),b(n))
 do i=1,n
   max=abs(a(i,i))
  do j=i+1,n
    q=abs(a(j,i))
   if( q > max ) then
     max = q
     k=j
   endif
  enddo
  if( max<eps ) then
    write(*,*) {'Матрица вырождена'}
    stop
  endif
  if( k /= i ) then
 do  j = i,m
  q=a(i,j)
  a(i,j)=a(k,j)
  a(k,j)=q
 enddo
  endif
 do i=m,i+1,-1
  a(i,j)=a(i,j)/a(i,i)
 enddo
 do l=i+1,n
  do j=i+1,n
    a(l,j)=a(l,j)-a(l,i)*a(i,j)
  enddo
 enddo
enddo
 do i=n,1,-1
   x(i-1+i0)=a(i,m)
  do l=i-1,1,-1
    a(l,m)=a(l,m)-a(l,i)*x(i-1+i0)
  enddo
 enddo
end subroutine  gauss1

PROGRAM  SolGauss1
```

```
      use sq
      open(4,file='sgauss.dat')
      open(5,file='sq.dat')
      read(4,91) n,((a(i,j),i=1,n),j=1,n)
  91 format(i7,f10.0)
      read(5,93) (b(i),i=1,n)
  93 format(f10.0)
      call gauss1()
      write(*,95) (b(i),x(i),i=1,n)
  95 format(2x,2f15.6)
    end
```

Программа 3.3 Метод Гаусса. 2 вариант

```
    program Gauss2
! Variables
      integer,parameter::n=4
      integer i,k
      real::a(n,n),a2(n,n),p(n),b2(n),b(n),x(n)=0.0,s
! Body of Gauss2
      a=reshape((/2., 3., 11., 5.,  &
                  1., 1., 5.,  2.,  &
                  2., 1., 3.,  2.,  &
                  1., 1., 3.,  4. /), &
                  shape=(/n,n/), order=(/2,1/))
      b=(/2.,1.,-3.,-3./)
      a2=a; b2=b
      do k = 1,n-1
        p(k+1:n)=a(k+1:n,k)/a(k,k)
       do i = k+1,n
         a(i,k+1:n)=a(i,k+1:n)-a(k,k+1:n)*p(i)
         b(i) = b(i)-b(k)*p(i)
       enddo
      enddo
      x(n) = b(n)/a(n,n)
      do k=n-1,1,-1
        s=sum(a(k,k+1:n)*x(k+1:n))
        x(k) = (b(k)-s)/a(k,k)
      enddo
      write(*,'(4f9.3)') x
```

```
  write(*,'(4f9.6)') b2-matmul(a2,x)    !невязка
end program Gauss2
```

Программа 3.2 Метод Гаусса. 3 вариант

```
program Gauss3
  use msimsl  !,nouse=>fact
  integer,parameter::ipath=1,n=3,Lda=n,Ldfact=n
  integer::ipvt(n),j
  real::a(Lda,Lda),fact(Ldfact,Ldfact),b(n),x(n)
  a=reshape((/ 1.0, 3.0, 3.0,   &
               1.0, 3.0, 4.0,   &
               1.0, 4.0, 3.0 /), &
               shape=(/Lda,Lda/), order=(/2,1/))
  b=(/1.0, -1.0, 2.5/)
  call Lftrg(n,a,Lda,fact,Ldfact,ipvt)
  call Lfsrg(n,fact,Ldfact,ipvt,b,ipath,x)
  write(*,*) x
end
```

3.4. Метод прогонки

Трехдиагональные системы линейных уравнений часто встречаются при численном решении систем дифференциальных уравнений в частных производных неявными методами – Глава 9, в задачах о сплайн – интерполяции (Глава 6 параграф 5).

Напомним, что трехдиагональная матрица, это такая матрица, у которой элементы расположены на главной диагонали, а также на диагоналях сверху и снизу от нее. Остальные элементы матрицы равны нулю. В методе прогонки используется это обстоятельство. К трехдиагональной матрице можно применить метод исключения Гаусса. Но при этом будет сделано много лишних операций. Метод прогонки гораздо более экономичен, чем метод Гаусса. Метод прогонки принадлежит так называемым прямым методам, как и метод Гаусса. В методе прогонки существует три этапа:

1. Приведение трехдиагональной матрицы к треугольной;
2. Задание рекурентного соотношения: $x_i = P_{i+1}x_{i+1} + Q_{i+1}$;
3. Осуществление обратного хода и определение всех неизвестных.

Систему алгебраических уравнений (3.7) можно записать в виде:

$$\begin{gathered} a_1 = c_n = 0, \\ a_i x_{i-1} - b_i x_i + c_i x_{i+1} = d_i, \quad i = 1, 2, \ldots, n. \end{gathered} \tag{3.15}$$

Приведем трехдиагональную матрицу к верхнетреугольной. Запишем (3.15) в виде системы уравнений.

$$\begin{array}{rcl} b_1 x_1 + c_1 x_2 + 0 + \ldots + 0 & = & d_1 \\ a_2 x_1 + b_2 x_2 + c_2 x_3 + 0 + \ldots & = & d_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots & & \\ 0 + \ldots + 0 + a_n x_{n-1} + b_n x_n & = & d_n \end{array} \tag{3.16}$$

Преобразуем первое уравнение системы (3.16) к виду:

$$x_1 = \alpha_1 x_2 + \beta_1, \tag{3.17}$$

здесь α и β называются коэффициентами прогонки и вычисляются по формулам: $\alpha_1 = -c_1/b_1$, $\beta_1 = \alpha_1/b_1$. При этом предполагается, что среди элементов, лежащих на главной диагонали отсутствуют нулевые элементы.

Подставив (3.17) во второе уравнение (3.16) и преобразовав его, получим: $x_2 = \alpha_2 x_3 + \beta_2$ и так далее.

На i-ом шаге:

$$x_i = \alpha_i x_{i+1} + \beta_i, \tag{3.18}$$

где

$$\begin{gathered} \alpha_i = -c_i/(b_i + a_i \alpha_{i-1}), \\ \beta_i = (d_i - a_i \beta_{i-1})/(b_i + a_i \alpha_{i-1}). \end{gathered} \tag{3.19}$$

На последнем шаге $x_{n-1} = \alpha_{n-1} x_n + \beta_{n-1}$.

Это прямой ход прогонки – вычисление прогоночных коэффициентов α_i, β_i. Как видно из (3.19), значения α_i, β_i вычисляются по известным коэффициентам a_i, b_i, c_i, d_i системы (3.16) и по уже вычисленным на предыдущем шаге значениям прогоночных коэффициентов α_{i-1}, β_{i-1}. Для начала прямого хода нам необходимо знать начальные значения коэффициентов α_1, β_1.

Обратный ход метода прогонки: находим:

$$x_n = \beta_n = (d_n - \alpha_n \beta_{n-1})/(b_n + a_n \alpha_{n-1}).$$

Все остальные значения x_i находятся из

$$x_i = \alpha_i x_{i+1} - \beta_i, \quad i = n-1, n-2, \ldots, 1.$$

Метод прогонки устойчив, если $|\alpha_i| \leq 1$ и корректен, если выполняются условия:

$$b_i + a_i \alpha_i \neq 0.$$

Покажем, что если выполняется условие устойчивости, то ошибки округления с течением времени не возрастают.

Пусть $\{\overline{x}_i, \overline{x}_{i+1}\}$ – возмущенное решение (то есть вычисленное с погрешностями). Допустим, что прогоночные коэффициенты α_i, β_i вычисляются точно. Тогда из (3.18)

$$\overline{x}_i - x_i = \alpha_{i+1}(\overline{x}_{i+1} - x_{i+1}).$$

Отсюда следует, что

$$\varepsilon_i = \overline{x}_i - x_i = \alpha_{i+1}\varepsilon_{i+1}.$$

Так как выполняется условие устойчивости $|\alpha_i| \leq 1$ отсюда следует, что погрешность с течением времени не возрастает.

Реализация метода прогонки требует $\sim O(n)$ операций. Напомним, что решение систем линейных уравнений методом исключения Гаусса требует $\sim O(n^3)$ операций.

Метод прогонки можно обобщить на случай, когда матрица 5-ти диагональная или вообще m - диагональная.

Пример 3.5. Решение трехдиагональной системы уравнений методом прогонки. Пусть у нас есть ленточная система уравнений:

$$\begin{cases} 2x_1 + x_2 + 0x_3 + 0x_4 &= -5 \\ x_1 + 10x_2 - 5x_3 + 0x_4 &= -18 \\ 0x_1 + x_2 - 5x_3 + 2x_4 &= -40 \\ 0x_1 + 0x_2 + x_3 + 4x_4 &= -27 \end{cases} \tag{3.20}$$

Прямой ход прогонки. Вычислим прогоночные коэффициенты, для этого обозначим:

$$\begin{aligned} \gamma_1 &= b_1 = 2, \quad \alpha_1 = -c_1/\gamma_1 = -1/2, \quad \beta_1 = d_1/\gamma_1 = -5/2, \\ \gamma_2 &= b_2 + a_2\alpha_1 = 10 + 1 \cdot (-1/2) = 19/2, \\ \alpha_2 &= -c_2/\gamma_2 = 10/10, \\ \beta_2 &= (d_2 - \alpha_2\beta_1)/\gamma_2 = (-18 - 1(-5/2))/(19/2) = -31/19. \\ \gamma_3 &= -85/19, \quad \alpha_3 = 38/85, \quad \beta_3 = 729/85, \\ \gamma_4 &= 375/85, \quad \beta_4 = -3024/378 = -8. \end{aligned}$$

Обратный ход прогонки:

$$x_4 = \beta_4 = -8, \quad x_3 = \alpha_3 x_4 + \beta_3 = 5,$$
$$x_2 = \alpha_2 x_3 + \beta_2 = 1, \quad x_1 = \alpha_1 x_2 + \beta_1 = -3.$$

Таким образом мы получили решение: $x_1 = -3, x_2 = 1, x_3 = 5, x_4 = -8$.

Программа 3.4 Программа матричной прогонки

```
    program Band_seq
      implicit none
! Variables
      integer,parameter::n=4
      integer::i
      real(4), dimension(n) :: b,d,x
      real(4),dimension(n-1):: a,c
      real(4):: arr(n,n)
      open(7,file='Band.dat')
! Body of Band_seq !  Задание ленточной матрицы

      arr = reshape((/ 1.0,  1.0,  0.0,  0.0,   &
                       2.0,  3.0, -1.0,  0.0,   &
                       0.0,  4.0,  2.0,  3.0,   &
                       0.0,  0.0,  2.0, -4.0   /), &
                       shape=(/n,n/), order=(/2,1/))
!  Задание столбца свободных членов
    d = (/7.,9.,10.,12./)
!  Определение векторов a,b,c
      do i = 1,n
        b(i) = arr(i,i)
      enddo
      do i = 1,n-1
        a(i) = arr(i+1,i)
        c(i) = arr(i,i+1)
      enddo
      call band(a,b,c,d,x,n)  ! Вызов подпрограммы метода прогонки.
     do i = 1,n
       write(*,91) i,x(i)
       write(7,91) i,x(i)
     enddo
 91  format(2x,'x(',i2,')= ',f12.6)
```

```
    end program Band_seq

    subroutine band(a,b,c,d,x,m)
      implicit none
      integer::m,j
      real(4), dimension(m)  :: b,d,x,temp
      real(4), dimension(m-1):: a,c
      real(4)                :: dummy
!   Последовательный алгоритм.

!   Подпрограмма для решения ленточных уравнений методом прогонки.
!   Состоит из прямого хода и  обратной подстановки. Размерность
!исходных массивов задается в главной программе. Вестор b
!содержит диагональные элементы  матрицы, a - элементы, лежащие
!под главной диагональю   c - элементы, лежащие над главной
!диагональю. Их размерность на  единицу меньше.   d - столбец
!свободных членов. Вектор x содержит  решение.

!  Body of Band
       dummy=b(1)
        x(1)=d(1)/dummy
!  Прямой ход метода прогонки
      do j=2,m
        temp(j)=c(j-1)/dummy
        dummy=b(j)-a(j-1)*temp(j)
         x(j)=(d(j)-a(j-1)*x(j-1))/dummy
      end do
!  Обратная подстановка
      do j=m-1,1,-1
        x(j)=x(j)-temp(j+1)*x(j+1)
      end do
     end subroutine band
```

Результаты работы программы матричной прогонки.

```
x( 1)=6.800000
x( 2)=0.200000
x( 3)=5.200000
x( 4)=-0.400000
```

3.5. Упражнения

Методом Гаусса найти численное решение систем линейных уравнений.

1.

$$\begin{aligned} 5x_1 + x_3 &= 11, \\ x_1 + 3x_2 - x_3 &= 4, \\ -3x_1 + 2x_2 + 10x_3 &= 6. \end{aligned}$$

2.

$$\begin{aligned} 3x_1 + x_2 - x_3 &= 7, \\ -2x_1 + 4x_2 + x_3 &= 5, \\ x_1 + x_2 + 3x_3 &= -3. \end{aligned}$$

3.

$$\begin{aligned} 5x_1 + x_2 - x_3 &= -5, \\ -1x_1 + 3x_2 + x_3 &= 51, \\ x_1 - 2x_2 + 4x_3 &= 1. \end{aligned}$$

Найти решение системы ленточных уравнений:

1.

$$\begin{aligned} x_1 + 2x_2 + 0 * x_3 + 0 * x_4 &= 7 \\ 2x_1 + 3x_2 - x_3 + 0 * x_4 &= 9 \\ 0 * x_1 + 0 * x_2 + 4x_2 + 2x_3 + 3x_4 &= 10 \\ 0 * x_1 + 0 * x_2 + 2x_3 - 4x_4 &= 12 \end{aligned}$$

2.

$$\begin{aligned} x_1 + x_2 + 0 * x_3 + 0 * x_4 &= 5 \\ 2x_1 - x_2 + 5x_3 + 0 * x_4 &= -9 \\ 0 * x_1 + 0 * x_2 + 3x_2 - 4x_3 + 2x_4 &= 19 \\ 0 * x_1 + 0 * x_2 + 2x_3 + 6x_4 &= 2 \end{aligned}$$

Найти решение следующих систем линейных уравнений методом простой итерации и с помощью итераций Гаусса – Зейделя:

1.

$$\begin{aligned} 4x_1 - x_2 + x_3 &= 7, \\ 4x_1 - 8x_2 + x_3 &= -21, \\ -2x_1 + x_2 + 5x_3 &= 15. \end{aligned}$$

2.

$$\begin{aligned} 4x_1 - x_2 &= 15, \\ x_1 + 5x_2 &= 9. \end{aligned}$$

3.

$$\begin{aligned} 5x_1 - x_2 + x_3 &= 10, \\ 2x_1 + 8x_2 - x_3 &= 11, \\ -x_1 + x_2 + 4x_3 &= 3. \end{aligned}$$

4.

$$\begin{aligned} 4x_1 - x_2 + x_3 &= 4, \\ x_1 + 6x_2 + 2x_3 &= 9, \\ -x_1 - 2x_2 + 5x_3 &= 2. \end{aligned}$$

Глава 4

Решение нелинейных уравнений

4.1. Введение

Нелинейные явления широко распространены в нашем мире. Они позволяют описывать сложные процессы, происхдящие в природе. Например, период полураспада радиоактивных элементов описывается уравнением:

$$M = M_0 e^{-kt}.$$

Здесь M – масса оставшегося к моменту времени t радиоактивного вещества, M_0 – начальная масса этого вещества, k период полураспада.

Другой пример. Рост популяции грызунов может быть описан уравнением:

$$P(t) = \alpha e^{k_1 t} - \beta t^{k_2} - \gamma t^{k_3}.$$

В этом уравнении P – количество особей в популяции к моменту времени t, $\alpha, \beta, \gamma, k_1, k_2, k_3$ – константы модели.

В общем случае рассмотрим нелинейное уравнение

$$f(x) = 0. \tag{4.1}$$

Здесь функция $f(x)$ определена и непрерывна на некотором множестве. На уравнение (4.1) могут быть наложены некоторые дополнительные условия – непрерывность k-ой производной, может потребоваться монотонность функции и так далее.

Требуется найти корни уравнения (4.1). Корни уравнения это такие числа $x_1, x_2, \ldots, x_n$, которые превращают уравнение (4.1) в тождество.

4.2. Решение нелинейных уравнений с одной независимой переменной

В этом параграфе будем изучать решения нелинейных уравнений с одной независимой переменной. Как известно из Главы 3, любую систему линейных уравнений можно решить за конечное количество арифметических операций. Кроме того, любая невырожденная система линейных уравнений имеет единственное решение. Для нелинейных уравнений это не всегда так. Вещественных решений может не быть совсем (рисунок 4.1):

$$f(x) = \cos(x) + 2 = 0 \tag{4.2}$$

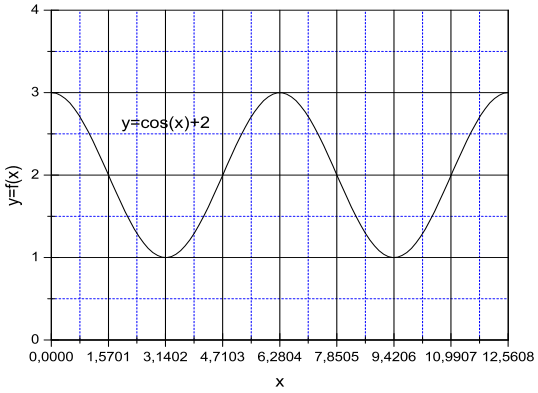

Рис. 4.1:

На рисунке 4.1 график функции (4.2) не пересекает ось X. С другой стороны вещественных корней может быть бесконечно много.

$$f(x) = \sin(x) = 0. \tag{4.3}$$

На рисунке 4.2 приведен график функции (4.3). Так как функция (4.3) периодическая, это означает, что она пересекает ось X бесконечное число раз.

Далеко не все нелинейные уравнения могут быть решены точно. Так, для полинома степени не выше четвертой, существуют формулы для его решения.

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e = 0.$$

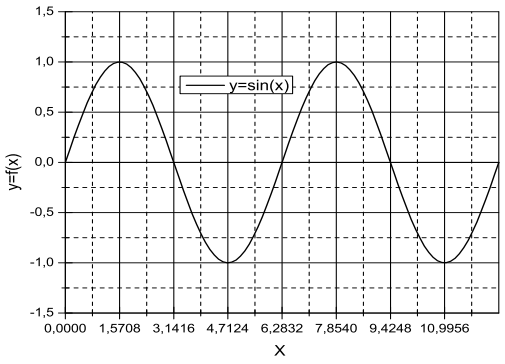

Рис. 4.2:

Однако, как показал Э. Галуа, для полиномов степени больше четырех в общем случае не существует формул для вычисления корней этого уравнения в радикалах.

Рассмотрим обобщенное понятие решения уравнения. Пусть x является корнем уравнения

$$f(x) = 0 \tag{4.4}$$

Пусть, кроме того, функция $f(x)$ непрерывна. Будем говорить, что $\overline{x}$ тоже является решением уравнения (4.4), если

$$|f(\overline{x})| \approx 0, \quad \text{или } |\overline{x} - x| \approx 0. \tag{4.5}$$

Второе выражение в (4.5) нуждается в пояснении. Функция $f(x)$ непрерывна по определению. Как уже обсуждалось во второй главе, на компьютере все вычисления выполняются с округлениями, и может не найтись такого числа x, чтобы точно выполнялось уравнение (4.4). Тогда, найденное в результате расчетов число $\overline{x}$ отличается от точного решения на ε_m. Покажем, что оба утверждения (4.5) эквивалентны. Допустим, что для функции $f'(x)$: $|f'(x)| \leq M$.

Разложим $f(x)$ в ряд Тейлора в окрестности точки $\overline{x}$:

$$|f(\widetilde{x})| = |f(\overline{x}) + f'(\xi)(\widetilde{x} - \overline{x})| = |f'(\xi)| \cdot |\widetilde{x} - \overline{x}| \leq M|\widetilde{x} - \overline{x}|. \tag{4.6}$$

Здесь $\xi \in [\widetilde{x}, \overline{x}]$. Поэтому из (4.6) следует, что если $|\widetilde{x} - \overline{x}| \approx 0 \Rightarrow |f'(\widetilde{x})| \approx 0$.

Процесс нахождения корня нелинейного уравнения – это, чаще всего, итерационный процесс. То есть, мы должны построить такую последовательность $x_1, x_2, \ldots, x_n$, что $f(x_1), f(x_2), \ldots, f(x_n) \to 0$. Процесс построения сходящейся последовательности $\{x_i\}$ обычно разделяют на два этапа:

- этап отделения корней,
- этап нахождения корня.

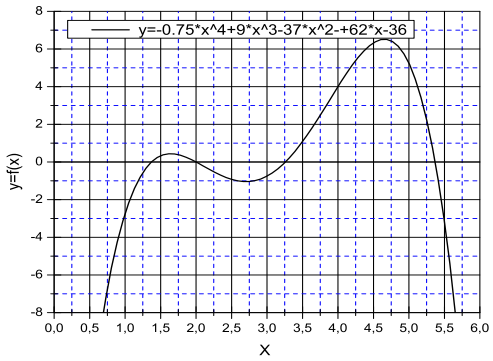

Рис. 4.3:

Отделение корня – это процедура нахождения такого промежутка $a \leq x \leq b$, на котором находится один и только один корень нелинейного уравнения (4.4).

Этап отделения корней не поддается строгой алгоритмизации. На этом этапе необходимо воспользоваться всей доступной информацией о поведении функции. Например, если известно, что $f(x)$ полином, можно воспользоваться правилом Декарта, а также теоремой Гюа и теоремой о среднем. В некоторых случаях можно воспользоваться информацией, связанной с физическими особенностями нелинейного уравнения. Часто используется метод визуализации, то есть сначала строится график уравнения (4.4), а затем, по нему определяется количество корней и отрезки $[a_i, b_i]$, на которых корень только один.

$$y = -0.75x^4 + 9x^3 - 37x^2 + 62x - 36 \tag{4.7}$$

На рисунке 4.3 приведен график функции (4.7), которая имеет четыре корня. Из него можно приблизительно найти интервалы, на которых находится только один корень.

Приведем формулировки теорем, которые могут помочь найти интервал отделения корней. Все эти теоремы доказываются в курсе математического анализа.

Теорема 4.1 *Пусть есть непрерывная функция $f(x)$, которая на концах отрезка $[a, b]$ принимает значения разных знаков. Тогда на отрезке $[a, b]$ существует по крайней мере один корень уравнения. Если же, кроме того, функция $f(x)$ дифференцируема, а ее производная сохраняет знак на $[a, b]$, то на этом отрезке находится только один корень уравнения.*

Следствие *Если на концах отрезка $[a, b]$ функция не меняет знаки, то на этом отрезке либо нет корней, либо их четное число.*

Корни x_j многочлена n-ой степени (в том числе и комплексные)

$$P_n(x) = \sum_{i=0}^{n} a_i x^i$$

находятся на интервале:

$$|x_j| \leq 1 + \frac{1}{a_n} \max[|a_0|, |a_1|, \ldots, |a_{n-1}|].$$

А по правилу знаков Декарта, разность между числом перемен знаков в последовательности $a_0, a_1, \ldots, a_n$ и числом положительных корней является либо положительным четным числом, либо нулем (для действительных корней).

Если заменить x на -x, то правило Декарта можно распространить и на отрицательные корни. Для отделения корней можно использовать и следующие теоремы:

Теорема 4.2 (Гюа). *Если все корни алгебраического уравнения являются действительными числами, то для последовательности коэффициентов $a_0, a_1, \ldots, a_n$ квадрат каждого не крайнего коэффициента больше произведения соседних с ним коэффициентов: $a_i^2 > a_{i-1} a_{i+1}$, $i = 2, 3, \ldots, n-1$.*

Теорема 4.3 *Если для каких-либо значений i $a_i^2 < a_{i-1} a_{i+1}$, то многочлен имеет по крайней мере одну пару комплексных корней.*

К основным численным методам нахождения корней нелинейных уравнений относятся простой итерации. метод половинного деления (дихотомии), метод Ньютона (касательных), метод секущих, метод Зейделя

и других. Эти методы различаются между собой сложностью, скоростью сходимости и надежностью.

4.2.1. Метод простой итерации

Рассмотрим задачу нахождения корня нелинейного уравнения (4.4) методом простой итерации. Итерации проводятся до тех пор, пока мы не найдем, с нужной точностью, корень уравнения. Будем считать, что мы ищем корень уравнения на отрезке $[a, b]$ и на этом отрезке есть только один корень уравнения.

Для начала процесса итерации нам необходимо задать:

1. Формулу для вычисления очередного шага $x_i = \varphi(x_{i-1})$,
2. Начальное значение x_0

Таким образом, необходимо построить последовательность $x_0, x_1, \ldots, x_n$ такую, что значения $\varphi(x_0), \varphi(x_1), \ldots, \varphi(x_n) \to x_0$, то есть корню уравнения.

Определение 4.1 *Неподвижной точкой функции $\varphi(x)$ называется действительное число Z, такое, что $Z = \varphi(Z)$.*

Геометрически неподвижная точка функции $y = f(x)$, это точки пересечения графиков функций $y = f(x)$ и $y = x$.

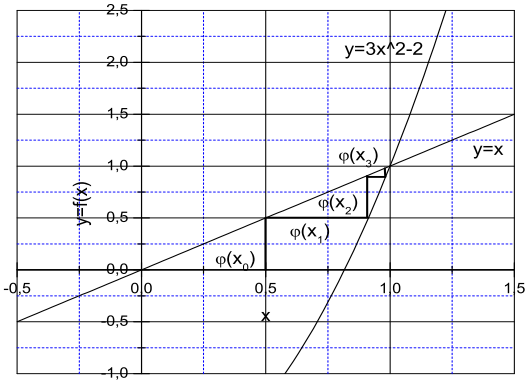

Рис. 4.4:

Определение 4.2 *Итерация $x_{i+1} = \varphi(x_i)$ для $i = 0, 1, \ldots, n$ называется итерацией неподвижной точки.*

Теорема 4.4 *Пусть $\varphi(x)$ – непрерывная функция и $\{x_i\}_{i=0}^{\infty}$ – последовательность, сгенерированная с помощью итерации неподвижной точки. Тогда, если существует $\lim\limits_{i\to\infty} x_i = x^*$, то x^* является неподвижной точкой.*

Геометрически можно представить метод простой итерации в виде (рисунок 4.4) отрезков прямой, которая последовательно сходится к пересечению линий $y = x$ и $y = 3x^2 - 2$.

Из начальной точки x_0 мы строим итерационный процесс:

$$x_{i+1} = \varphi(x_i), \tag{4.8}$$

который, при правильном выборе начальной точки сходится к корню. В этом методе важно правильно выбрать начальную точку. Если в качестве начального значения выбрать точку $x_0 = 1.2$, то мы получим расходящееся решение.

Сходимость метода. Пусть $\varphi(x)$ непрерывна и имеет непрерывную первую производную. Тогда из теоремы следует:

$$x_{i+1} - x^* = \varphi(x_i) - \varphi(x^*) = (x_i - x^*)\varphi'(\xi)$$

где $x_i \le \xi \le x^*$. Отсюда следует, что если $|\varphi'(x)| \le q < 1$, то размер отрезков $\{x_i - x^*\}$ убывает. Таким образом, это условие является достаточным условием сходимости.

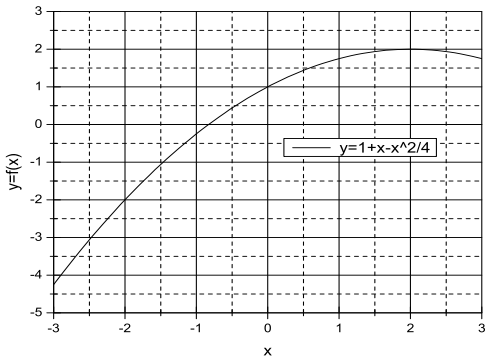

Рис. 4.5:

Пример 4.1. Рассмотрим функцию

$$\varphi(x) = 1 + x - x^2/4. \tag{4.9}$$

Производная $\varphi'(x) = 1 - x/2$. Неподвижные точки можно найти, решив уравнение

$$x = \varphi(x). \tag{4.10}$$

В данном случае для уравнение $\varphi(x) = 1 + x - x^2/4$ получим, используя (4.10), что $x^2 = 4$ имеет две неподвижные точки x=-2 и x=2. На рисунке 4.5 приведен график функции (4.9).

Так как $\varphi'(x) \geq 3/2$ на отрезке $[-3, -1]$ последовательность не сходится к неподвижной точке -2: $x_0 = -2.05, x_1 = -2.100625,\ x_2 = -2.203781,\ x_3 = -2.417943,\ x_4 = -2.879556,\ x_5 = -3.952517,\ \lim\limits_{i\to\infty} x_i = -\infty$.

Если построить такую же последовательность в окрестности второй неподвижной точки $\varphi'(x) < 1/2$, последовательность сходится к x=2: $x_0 = 1.6,\ x_1 = 1.96, x_2 = 1.9996,\ x_3 = 2.0 \Rightarrow \lim\limits_{i\to\infty} x_i = 2$.

4.2.2. Метод дихотомии (деления отрезка пополам)

Одним из простых и надежных методов является метод дихотомии или метод деления отрезка пополам. Для работы с этим методом необходимо, чтобы нелинейная функция $f(x) = 0$ на отрезке $[a, b]$ имела только один корень, на концах которого функция бы имела разные знаки –

$$f(a) \cdot f(b) < 0, \quad a < b. \tag{4.11}$$

Кроме того, необходимо, чтобы функция $f(x)$ была непрерывна на отрезке [a,b]. Метод половинного деления позволяет построить такую последовательность вложенных отрезков $[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]$, которая сходится к корню уравнения. На каждом шаге отрезок [a,b] делится пополам, из этих двух отрезков выбирается тот, для которого выполняется условие (4.11). Такие шаги выполняются до тех пор, пока $|f(x_i)| < \varepsilon$, либо $|x_i - x_{i-1}| < \delta$.

1. Если $|x_i - x_{i-1}| < \delta$ корень найден.
2. Вычисляем $\xi = 1/2 \cdot (x_i + x_{i-1})$. Находим $f(\xi)$. Если $|f(\xi)| < \varepsilon$ – корень найден.

3. Если $f(x_i)f(\xi) < 0$, то $x_{i-1} = \xi$, иначе $x_i = \xi$. Перейти к 1.

Сходимость метода. На каждом шаге итерации мы уменьшаем отрезок в два раза. Пусть e_i – ошибка на i - ой итерации.

$$e_i = \xi - \overline{x}, \text{ где } \overline{x} \text{ корень}, \frac{e_{i+1}}{e_i} \approx \frac{1}{2}.$$

Скорость сходимости r определяется так.

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i|^r} = c, \tag{4.12}$$

где c – отличная от нуля константа. $|e_{i+1}| = O(|e_i|^r)$. Если r=1, то скорость сходимости называтся линейной, если r>1 – сверхлинейной [1], если r=2 – квадратичной. Скорость сходимости зависит от r и от c. Если скорость сходимости линейна, а $c \geq 1$, то метод дихотомии может и не сходится. Несмотря на то, что большие значения r в конечном счете обеспечивают более быструю сходимость, линейные скорости сходимости будут вполне удовлетворительными, если константа c достаточно мала.

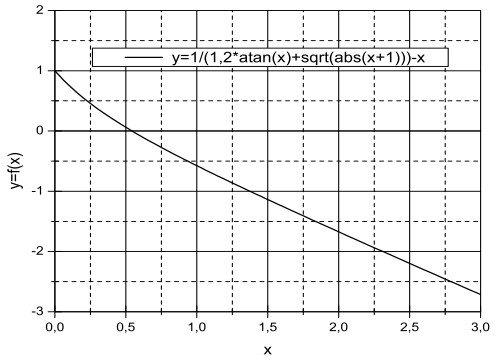

Рис. 4.6:

Можно сделать вывод, что метод дихотомии надежен и прост, почти всегда сходится и не порождает ошибок округления.

Пример 4.2 Найти корень уравнения

$$y = 1/(1.2 \operatorname{arctg}(x) + \sqrt{|x+1|}) - x \tag{4.13}$$

на отрезке $[0,3]$ методом деления отрезка пополам. На рисунке 4.6 представлен график функции 4.13 на отрезке $[0,3]$. Ниже приведена таблица, полученная по программе 4.1.

i	Точное значение x_i
1	0.7500000
2	0.3750000
3	0.5625000
4	0.4687500
5	0.5156250
6	0.5390625
7	0.5507812
8	0.5449219
9	0.5419922
10	0.5434570
11	0.5441895
12	0.5438232
13	0.5436401
14	0.5435486
15	0.5435028

В программе 4.1 задаются начальные значения a, b, eps. Отрезок a, b делится пополам. Затем происходит проверка того, на каком отрезке функция имеет разные знаки. Этот отрезок снова делится пополам и так до тех пор, пока не будет, с нужной точностью, найден корень уравнения.

Программа 4.1 Метод деления отрезка пополам.

```
   program Dichotomia
    implicit none
! Объявление переменных и констант
    real(4)::a=0.,b=3.,c,eps=0.0001,delta=0.0001,fc,fa,fb,fx,f
    integer::iter=0,j
! Body of Dichotomia
    open(7,file='dich.dat')
     fa = f(a)
     fb = f(b)
     do while(abs(a-b) >= eps )
       iter = iter+1
       c = (a+b)/2.
       fc = f(c)
```

```
  if( fc*fa < 0. ) then
     b = c
    fb = fc
  elseif( fc*fb < 0. ) then
   a = c
    fa = fc
  endif
  write(7,*) 'iter = ',iter,' x = ',(a+b)/2.
 if( f((a+b)/2.) < eps ) exit
 enddo
 write(*,*) 'iter = ',iter,' x = ',(a+b)/2.
end program Dichotomia

real function f(x)
  f = 1./(1.2*atan(x)+sqrt(abs(x+1.)))-x
end
```

4.2.3. Метод Ньютона

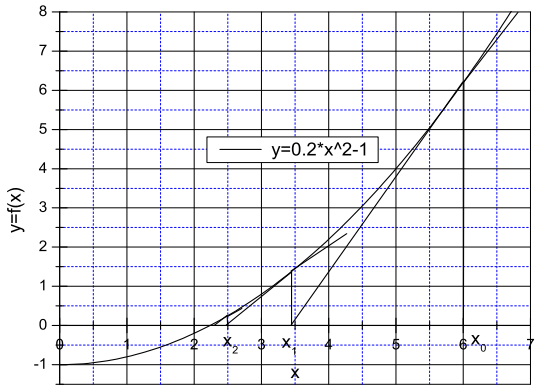

Рис. 4.7:

Метод Ньютона является одним из лучших методов решения нелинейных уравнений. Для первого шага нам необходимо найти начальное приближение x_0. Алгоритм метода Ньютона заключается в следующем: мы строим касательную к исходной функции $f(x)$ в начальной точке x_0. Точка пересечения касательной x_1 с осью X принимается за следующее приближение, и так до тех пор, пока не будет найден, с заданной точностью,

корень уравнения. На рисунке 4.7 приведены несколько последовательных итераций по методу Ньютона.

Для того, чтобы вывести формулы метода Ньютона, разложим функцию $f(x)$ в ряд Тейлора в окрестности точки x_i:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \dots$$

Касательная в точке x_i задается при помощи двух первых членов разложения:

$$f(x_i) + f'(x_i)(x - x_i) = 0$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.14}$$

Применим метод Ньютона для решения уравнения $f(x) = x^2 - 2 = 0$. Пусть $x_0 = 1$, $f'(x) = 2x$.

1. Первая итерация: $x_1 = x_0 - \frac{x_0^2-2}{2x_0} = 1 - (1-2)/2 = 3/2 = 1.5$.
2. Вторая итерация: $x_2 = 3/2 - (9/4 - 2)/3 = 17/12 = 1.416667$.
3. Третья итерация: $x_3 = 17/12 - (289/144 - 2)/(17/6) = 577/408 \approx 1.414216$.

Сравним x, полученный после третьей итерации, со значением квадратного корня из 2 $\overline{x} = \sqrt{2} \cong 1.4142214$. отметим, что расхождение начинается в пятом знаке после запятой.

Метод Ньютона имеет второй порядок сходимости: $r = 2$. Однако, у метода Ньютона есть и недостатки. Если на очередной итерации $f'(x_i) = 0$, то возникает неопределенность. Если $f'(x_i) \approx 0$, то появляются проблемы, связанные с тем, что новое приближение x_{i+1} может стать значительно хуже, чем предыдущее приближение x_i. Вторым недостатком метода Ньютона является необходимость вычисления $f'(x_i)$. Вычисление производной может оказаться сложной задачей, которая потребует много времени, а иногда, когда в $f(x_i)$ входят интегралы или дифференциальные уравнения и вовсе невозможной.

Программа 4.2 Метод Ньютона.

```
program Newton
  implicit none
  real:: x,x1,eps=1.0e-06,fx,f,f1
```

```
      integer::i=0
      x1=1.0
     do
        fx = f(x1)
       if( abs(fx)<eps ) exit
       if( f1(x1) != 0 ) x1=x1-f(x1)/f1(x1)
        i=i+1
        write(*,91) i,x1
     enddo
      write(*,91) i,sqrt(2.0)
 91   format(2x,' i = ',i4,2x' x = ',f15.8)
    end program Newton
    real function f(x)
      f=x*x-2.0
    end

    real function f1(x)
      f1=2.0*x
    end
```

Найдем корень нелинейного уравнения $x^2 - 2 = 0$ методом Ньютона. В таблице приведены последовательные итерации метода.

i	x_i
1	1.5000000
2	1.4166666
3	1.4142156
4	1.4142135

Значение корня с семью значащими цифрами равно x=1.4142135

Это простейшая программа на Фортране 90, реализующая метод Ньютона. Однако эта программа не застрахована от нескольких осложнений. Во-первых у нас нет уверенности, что очередная итерация приближает нас к решению. Во-вторых очередной шаг может быть слишком большим. Для того, чтобы предотвратить слишком большие шаги, наложим ограничение на очередной шаг:

$$|s| \leq \delta,$$

δ – некоторое положительное число, которое задает ограничение на шаг.

Для устранения некоторых ограничений метода Ньютона рассмотрим его модификацию. Для этого, вместо уравнения (4.14) используем уравнение

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_0)} \tag{4.15}$$

Использование (4.15) вместо (4.14) приводит к тому, что угол наклона касательной будет одинаковым на каждом шаге. Эта модификация позволяет сократить количество вычислений производных. Но, с другой стороны, скорость сходимости этого метода не будет квадратичной, то есть r<2.

Если вычисление производной затруднено, можно заменить производную ее конечно–разностной аппроксимацией:

$$f'(x) = \frac{f(x_i + h) - f(x_i - h)}{2h} \tag{4.16}$$

Величина h выбирается таким образом, чтобы сбалансировать ошибку округления и ошибку аппроксимации

$$h = \sqrt{\frac{3|f(x_k)|\varepsilon_m}{|f'''(x_k)|}}.$$

Для большинства вычислений можно принять $h = \sqrt[3]{\varepsilon_m}$.

Применение центральной разности (4.16), вместо правой или левой разности позволяет избежать потери точности в том случае, если $|f'(x_i)|$ мало.

Упражнения.

- Измените программу 4.2 таким образом, чтобы вместо $f'(x_i)$ использовать уравнение (4.16).
- Что произойдет, если метод деления пополам использовать для функции $f(x) = 1/(x-2)$:
  1. На интервале [3,7]?
  2. На интервале [1,7]?
- Вычислите корень уравнения
  1. $\ln(x) - 5 + x = 0$, на интервале $[3.2, 4.0]$
  2. $x^2 - 10x + 23 = 0$, на интервале $[6.0, 6.8]$
  3. $e^x - 2 - x = 0$, на интервале $[-2.4, -1.6]$
  4. $\cos(x) + 1 - x = 0$, на интервале $[0.8, 1.6]$

4.2.4. Метод секущих (хорд)

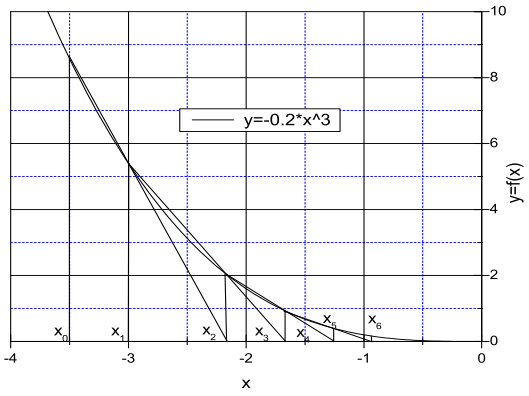

Рис. 4.8:

Если вычисление производной, которая используется в методе Ньютона затруднено или невозможно, можно заменить касательную на секущую. В методе секущих необходимо задать два последовательных приближения x_0, x_1. Идея метода состоит в том, что через эти две точки проводится секущая и значение x_2, в котором она пересекает ось x, берется в качестве нового приближения:

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} \tag{4.17}$$

Применение секущей вместо касательной приводит к замедлению скорости сходимости. Можно показать, что метод секущих сходится сверхлинейно, то есть со скоростью

$$r = 1/2(1 + \sqrt{5}) \approx 1.618.$$

Константа c равна

$$c = \left| \frac{f''(\overline{x})}{2f'(\overline{x})} \right|^{1/2}$$

На рисунке 4.8 приведены последовательные итерации, полученные по методу секущих. Недостатки метода секущих сходны с недостатками метода Ньютона. Если на очередном шаге $f(x_i) \approx f(x_{i-1})$ – метод становится неопределенным.

Пример 4.3 Найдем корень уравнения

$$x\sin(x) - 1 = 0. \tag{4.18}$$

на отрезке [0,2], (a=0, b=2). На рисунке 4.9 приведен график функции (4.18). Из (4.17) найдем:

$$x_1 = 2 - \frac{0.81859485(2-0)}{0.81859485 - (-1)} = 1.09975017,$$

$$f(x_1) = -0.02001921.$$

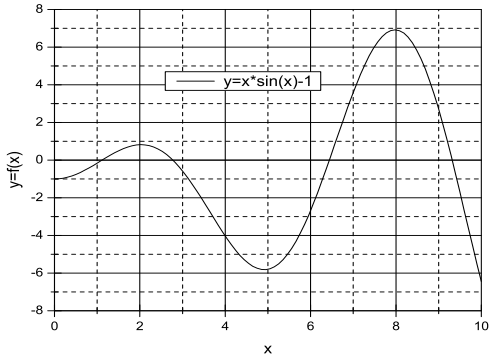

Рис. 4.9:

На интервале [1.09975017,2] функция меняет знак. Поэтому оставляем вместо a новое значение 1.09975017. По формуле (4.17) находим новые значения: $x_2 = 1.12124074$ и $f(x_2) = 0.00983461$. После четвертой итерации находим значение x=1.11416120.

Программа 4.3 Метод секущих

```
program Secant
  implicit none
   real::x,x0,x1,f,fx,fx0,fx1,eps=1.0e-06
   integer::n=0
   x0 = 1.0
   x1=2.0
   fx0=f(x0)
   fx1=f(x1)
```

```
    do while( abs(0.5*(fx0+fx1)) > eps )
       x = x1-fx1*(x1-x0)/(fx1-fx0)
       fx=f(x)
       n=n+1
     if( fx0*fx < 0 ) then
       x0=x; fx0=fx
     else
       x1=x; fx1=fx
     endif
    write(*,91) n,x,fx
    enddo
    write(*,91) n,x,fx
 91 format(2x,'i=',i3,'x = ',f15.7,2x,'f(x0) = ',f15.7 )
    end program Secant

    real function f(x)
      f = x**3+x-3.
    end
```

Решим, используя программу Secant нелинейное уравнение $x^2 - 2 = 0$. В таблице приведены восемь последовательных итераций метода секущих, полученных по данной программе.

i	x_i
1	1.1250000
2	1.2277580
3	1.2125341
4	1.2134657
5	1.2134084
6	1.2134118
7	1.2134117

4.2.5. Метод Мюллера

Дальнейшим развитием метода секущих является метод Мюллера. Вместо прямой, которая применяется в методе секущих, в методе Мюллера используется парабола, которая строится по трем ранее найденным точкам $x^{(k-2)}, x^{(k-1)}, x^{(k)}$. Этот метод является трехшаговым, так как для нахождения очередной итерации нам необходимо знать три предыдущие.

Необходимым условием построения полинома второй степения является несовпадение значений $f(x^{(k-2)}), f(x^{(k-1)}), f(x^{(k)})$. Это выполняется, если функция $y = f(x)$ на исследуемом отрезке строго монотонна. В окрестности простого (не кратного) корня метод Мюллера сходится быстрее метода секущих и по скорости сходимости приближается к методу Ньютона. Метод Мюллера позволяет найти как вещественные, так и комплексные нули функции. Используя тип COMPLEX в Fortran метод Мюллера легко запрограммировать для нахождения комплексных корней.

Рассмотрим квадратный полином:

$$P(y) = ay^2 + by + c.$$

Коэффициенты этого уравнения можно получить, решая систему линейных уравнений:

$$\begin{array}{rcl} a\left(y^{(k-2)}\right)^2 + by^{(k-2)} + c & = & x^{(k-2)}, \\ a\left(y^{(k-1)}\right)^2 + by^{(k-1)} + c & = & x^{(k-1)}, \\ a\left(y^{(k)}\right)^2 + by^{(k)} + c & = & x^{(k)}. \end{array} \tag{4.19}$$

В уравнении (4.20) используется обозначение: $y^{(i)} = f(x^{(i)}),\ i = k-2, k-1, k$.

Свободный член $P(y)$ можно найти по правилу Крамера: $e = D_3/D$, где

$$D = \begin{vmatrix} \left(y^{(k-2)}\right)^2 & y^{(k-2)} & 1 \\ \left(y^{(k-1)}\right)^2 & y^{(k-1)} & 1 \\ \left(y^{(k)}\right)^2 & y^{(k)} & 1 \end{vmatrix},$$

$$D_3 = \begin{vmatrix} \left(y^{(k-2)}\right)^2 & y^{(k-2)} & x^{(k-2)} \\ \left(y^{(k-1)}\right)^2 & y^{(k-1)} & x^{(k-1)} \\ \left(y^{(k)}\right)^2 & y^{(k)} & x^{(k)} \end{vmatrix}.$$

Отсюда:

$$\begin{aligned} x^{(k+1)} = P(0) = x^{(k-2)} & \frac{y^{(k-1)}y^{(k)}}{\left(y^{(k-2)} - y^{(k-1)}\right)\left(y^{(k-2)} - y^{(k)}\right)} + \\ + x^{(k-1)} & \frac{y^{(k-2)}y^{(k)}}{\left(y^{(k-1)} - y^{(k-2)}\right)\left(y^{(k-1)} - y^{(k)}\right)} + \\ + x^{(k)} & \frac{y^{(k-2)}y^{(k-1)}}{\left(y^{(k)} - y^{(k-2)}\right)\left(y^{(k)} - y^{(k-1)}\right)}. \end{aligned}$$

Пример 4.4 Найти корень функции

$$y = x^3 - e^x. \tag{4.20}$$

методом Мюллера. Начальное приближение $x^{(0)} = 5.0,\ x^{(1)} = 5.2,\ x^{(2)} = 5.4$.

Программа метода Мюллера 4.4

```
    program Muller
      implicit none
! Variables
      real::x,x0,x1,x2,eps=1.0e-06
      real,external::f
      integer::imax=25
      logical root
! Body of Muller
        x0=5.0; x1=5.2; x2=5.4
     if( root(f,x0,x1,x2,eps,imax,x) ) &
         write(*,*) 'x = ',x,' f(x) = ',f(x)
    end program Muller

   logical function root(f,x0,x1,x2,eps,imax,x)
      real::f,x0,x1,x2,eps,x,y0,y1,y2,t01,t02,t12
      integer:: imax,k
      k=0
      y0=f(x0); y1=f(x1); y2=f(x2)
      t01=y0-y1; t02=y0-y2; t12=y1-y2
     if( check0() ) return
      x=x0*(y1/t01)*(y2/t02)+x1*(y0/t01)*(y2/t12)+ &
        x2*(y0/t02)*(y1/t12)
    do while( abs(x-x2) > eps )
      k = k+1
     if( k>imax ) exit
      x0=x1; x1=x2; x2=x
      y0=y1; y1=y2; y2=f(x2)
      t01=y0-y1; t02=y0-y2; t12=y1-y2
     if( check0() ) return
      x=x0*(y1/t01)*(y2/t02)-x1*(y0/t01)*(y2/t12)+ &
        x2*(y0/t02)*(y1/t12)
    enddo
```

```
  root=k<=imax
 if( .not.root ) then
  write(*,*) 'Roots not found'
 endif
contains
  logical function check0()
    check0 = abs(t01)<tiny(t01).or.abs(t02)<  &
    tiny(t01).or.abs(t12)<tiny(t01)
   if( check0 ) then
    write(*,*) 'Dividing by zero'
    root = .false.
   endif
  end function check0
 end function root

 function f(x)
    real::f,x
    f=x**3-exp(x)
  end
```

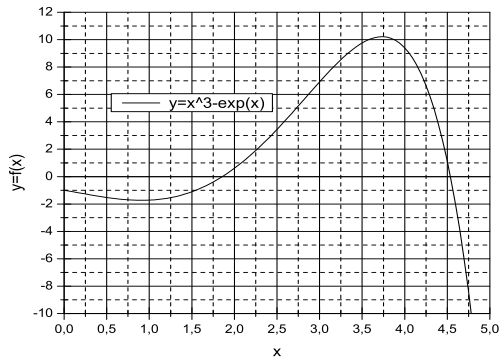

Рис. 4.10:

В данной программе используется встроенная функция tiny(x), которая возвращает наименьшее число, которое соответствует типу данных. Параметр x должен быть вещественного типа. На рисунке 4.10 приведен график функции (4.20). Корень $x = 4.536404$.

4.3. Решение систем нелинейных уравнений

Рассмотрим задачу решения систем нелинейных уравнений:

$$\begin{array}{l} f_1(x_1, x_2, \ldots, f_n) = 0, \\ f_2(x_1, x_2, \ldots, f_n) = 0, \\ \ldots\ldots\ldots\ldots \\ f_n(x_1, x_2, \ldots, f_n) = 0. \end{array} \tag{4.21}$$

Введем векторы $\overline{x} = \{x_1, x_2, \ldots, x_n\}$, $f(x) = \{f_1(\overline{x}), f_2(\overline{x}), \ldots, f_n(\overline{x})\}$. Тогда, с помощью введенных векторов можно записать систему нелинейных уравнений в виде:

$$f(\overline{x}) = 0.$$

Для нахождения корней нелинейных уравнений (4.21) используются методы простой итерации, Зейделя, Ньютона и другие методы.

4.3.1. Метод простой итерации

Метод простой итерации можно применить для решения системы нелинейных уравнений. Приведем систему (4.21) к виду:

$$\begin{cases} x_1 = \varphi_1(x_1, x_2, \ldots, x_n) \\ x_2 = \varphi_2(x_1, x_2, \ldots, x_n) \\ \ldots\ldots\ldots, \\ x_n = \varphi_n(x_1, x_2, \ldots, x_n). \end{cases} \tag{4.22}$$

Для того, чтобы решить систему нелинейных уравнений методом простых итераций, необходимо задать начальное приближение:

$$x^0 = \{x_1^0, x_2^0, \ldots, x_n^0\}.$$

Используя это начальное приближение построим итерационную последовательность, которая позволяет получить корни уравнения (4.22):

$$\begin{array}{rcl} x_1^{(k+1)} & = & \varphi_1(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), \\ x_2^{(k+1)} & = & \varphi_2(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), \\ \ldots\ldots & & \\ x_n^{(k+1)} & = & \varphi_n(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), \end{array}$$

здесь(k) – номер итерации.

Метод простой итерации сходится, если выполняется следующая теорема.

Теорема 4.5 *Пусть функции* $\varphi_i(x)$ *и* $\varphi_i'(x)$, $i = 1, 2, ..., n$, *непрерывны в некоторой области* Γ, *причем выполняется неравенство*

$$\max_{x\in\Gamma} \max_i \sum_{j=1}^{n} \left| \frac{\partial \varphi_i(x)}{\partial x_j} \right| \leqslant 1,$$

где q *константа. Если последовательные приближения* $x^{k+1} = \Phi(x^k)$, $(k = 0, 1, ...)$ *не выходят из области* Γ, *процесс сходится, при этом* $\overline{x} = \lim\limits_{k\to\infty} x^{(k)}$ *и вектор* $\overline{x}$ *является единственным решением системы (4.22).*

Приведем пример использования метода простой итерации.

Пример 4.5 Методом простой итерации решить систему нелинейных уравнений:

$$\begin{aligned} x_1^2 + x_2^2 &= 9, \\ x_1 + x_2 &= 3, \\ x_1 + x_2 + 3x_3 &= 6. \end{aligned} \tag{4.23}$$

Ниже приведена программа для решения системы нелинейных уравнений методом итераций.

Программа 4.5

```
program iteration_method
 implicit none
 integer::nmax=25,k=0
 real::x1,x2,x3,x11,x22,x33,eps=1.0e-06,f1,f2,f3
 open(7,file='iter.dat')
 x1=0.5
 x2=0.5
 x3=0.5
do
  k=k+1
  x11 = f1(0.0,x2,x3)
  x22 = f2(x1,0.0,x3)
  x33 = f3(x1,x2,0.0)
 if( abs(x11-x1)+abs(x22-x2)+abs(x33-x3) < eps ) then
   exit
 elseif( k > nmax ) then
```

```
        exit
      endif
       x1 = x11
       x2 = x22
       x3 = x33
       write(7,91) k,x1,x2,x3
       write(*,91) k,x1,x2,x3
     enddo
 91  format(2x,'i = ',i3,2x,' x1 = ',f15.7,2x, &
     'x2 = ',f15.7,2x,'x3 = ',f15.7)
    end program iteration_method

    real function f1(x1,x2,x3)
      real:: x1,x2,x3
      f1 = 3.0-x2
    end

    real function f2(x1,x2,x3)
      real:: x1,x2,x3
      f2 = sqrt(9.0-x1*x1)
    end

    real function f3(x1,x2,x3)
      real:: x1,x2,x3
      f3 = (6.0-x1-x2)/3.0
    end
```

Решение системы уравнений (4.23) с заданной точностью $\varepsilon = 10^{-6}$ найдено за 11 итераций:

```
i =  11   x1 = 0.0000000 x2 = 3.0000000  x3 = 1.0000000
```

4.3.2. Метод Зейделя

Модификацией метода простых итераций служит метод Зейделя. Для ускорения процесса вычисления корней уравнений можно воспользоваться уже вычисленными на предыдущей итерации значениями. (Таким подходом мы пользовались в предыдущей главе.) Для решения систем нелинейных уравнений необходимо задать начальное приближение $x^{(0)}$ и

точность, с которой мы хотим найти решение $\varepsilon > 0$. При вычислении очередной $x_1^{(k+1)}$ итерации для первого уравнения системы мы пользуемся значениями $x^{(k)}$, найденными на предыдущей итерации. При вычислении $x_2^{(k+1)}$, $x_3^{(k+1)}$ и последующих можно воспользоваться уже вычисленными на этой итерации значениями:

$$\begin{aligned} x_1^{(k+1)} &= \varphi(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}), \\ x_2^{(k+1)} &= \varphi(x_1^{(k+1)}, x_2^{(k)}, \dots, x_n^{(k)}), \\ x_3^{(k+1)} &= \varphi(x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_n^{(k)}), \\ \dots &= \dots \\ x_n^{(k+1)} &= \varphi(x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_n^{(k+1)}). \end{aligned} \tag{4.24}$$

Метод Зейделя позволяет существенно ускорить процесс вычисления корней системы нелинейных уравнений. Если переделать программу 4.5 в соответствии с (4.24), то для решения системы уравнений (4.23) потребуется всего 6 итераций. Таким образом, использование метода Зейделя позволяет почти в два раза сократить количество итераций, необходимых для нахождения корней с требуемой точностью. Это ускорение связано с тем, что мы, на каждой итерации начиная со второй, используем уже вычисленные значения.

```
i = 6   x1 = 0.0000000  x2 = 3.0000000  x3 = 1.0000000
```

4.3.3. Метод Ньютона для решения систем нелинейных уравнений

Будем искать решение $\overline{x} = \{x_1, x_2, \dots, x_n\}$ системы уравнений (4.21). Аналитическое решение такой системы удается найти очень редко. В таких случаях обычно ищется приближенное решение, которое отличается от точного на заранее заданное значение точности ε. В основу метода Ньютона для решения систем нелинейных уравнений, как и в случае одного уравнения, положен итеративный алгоритм. То есть, если найдено приближение $x^{(k)}$ к решению системы $\overline{x}$, то следующее приближение будет: $x^{(k+1)} = \psi(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$.

Разложим систему (4.21) в ряд Тейлора в точке $x^{(k)}$, и ограничимся

только линейными членами разложения:

$$\begin{array}{l}
\frac{\partial f_1}{\partial x_1}\triangle x_1^{(k)} + \frac{\partial f_1}{\partial x_2}\triangle x_2^{(k)} + \ldots + \frac{\partial f_1}{\partial x_n}\triangle x_n^{(k)} = -f_1(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), \\
\frac{\partial f_2}{\partial x_1}\triangle x_1^{(k)} + \frac{\partial f_2}{\partial x_2}\triangle x_2^{(k)} + \ldots + \frac{\partial f_2}{\partial x_n}\triangle x_n^{(k)} = -f_2(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}), \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\frac{\partial f_n}{\partial x_1}\triangle x_1^{(k)} + \frac{\partial f_n}{\partial x_2}\triangle x_2^{(k)} + \ldots + \frac{\partial f_n}{\partial x_n}\triangle x_n^{(k)} = -f_n(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}).
\end{array} \tag{4.25}$$

Матрица

$$J(x^{(k)}) = \begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\
\cdots & \cdots & \cdots & \cdots \\
\frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n}
\end{pmatrix} \tag{4.26}$$

коэффициентов системы (4.25) называется матрицей Якоби.

Достаточное условие сходимости метода Ньютона дается следующей теоремой. Пусть $W(x,r) = \{x \in R^n : |\overline{x} - x| < r\}$ – открытая окрестность радиуса r с центром в точке x. $J(x) \in L_\gamma(W(x_*, r))$ - означает, что якобиан преобразования J непрерывен по Липшицу: $\|J(y) - J(x)\| \leq \gamma\|y - x\|,\ \forall x, y \in W(x_*, r)$. Здесь γ – константа Липшица.

Теорема 4.6 *Пусть функция F(x) непрерывно дифференциируема в открытом выпуклом множестве $\Gamma \in R^n$. Кроме того, пусть существует $x \in R^n$ и $r, \beta > 0$ такие, что $W(x_*, r) \subset \Gamma$, $F(x_*) = 0$ и существует $J^{-1}(x_*)$, причем $\|J^{-1}(x_*)\| \leq \beta$ и $(x) \in L_\gamma(W(x_*, r))$. Тогда существует $\varepsilon > 0$ такое, что для всех $x^0 \in W(x_*, \varepsilon)$ последовательность $x^1, x^2, .., x^n$, порождаемая соотношением (4.21) сходится к x_* и удовлетворяет неравенству*

$$\|x^{(k+1)} - x_*\| \leq \beta\gamma\|x^{(k)} - x_*\|^2, \quad k = 0, 1, ...$$

Из теоремы следует квадратичная скорость сходимости.

Алгоритм решения системы нелинейных уравнений методом Ньютона аналогичен (4.14):

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})F(x^{(k)}) \tag{4.27}$$

Решим систему (4.27) относительно $\triangle x_i^{(k)},\ k = 1, 2, \ldots, n$, где

$$\triangle x_i^{(k)} = x_i^{(k+1)} - x_i^{(k)} = -J^{-1}(x^{(k)})F(x^{(k)}).$$

Умножив это выражение на матрицу Якоби слева, и используя (4.25) получим систему линейных алгебраических уравнений

$$J(x^{(k)})\triangle x^{(k)} = -J(x^{(k)})J^{-1}(x^{(k)})F(x^{(k)}) = -F(x^k). \qquad (4.28)$$

Полученную систему уравнений будем решать методом Гаусса. Критерием завершения процедуры нахождения корней служит условие:

$$||x^{(k+1)} - x^{(k)}|| < \varepsilon. \qquad (4.29)$$

Таким образом алгоритм решения системы нелинейных уравнений методом Ньютона будет выглядеть следующим образом.

1. Зададим начальное приближение $\overline{x}^{(0)}$ и точность ε.
2. Из (4.28) найдем $\triangle x_i^{(0)}$, $i = 1, 2$. Для решения полученной системы линейных уравнений используем метод Гаусса.
3. Найдем следующее значение $\overline{x}^{(1)}$.
4. Найдем $\delta = \max\limits_i\{|\triangle x_i^{(0)}|\}$. Если $\delta > \varepsilon$ переходим к пункту 1, в противном случае мы нашли корни нелинейной системы уравнений.

Пример 4.6 Решить систему нелинейных уравнений:

$$\begin{aligned} x_1^2 + x_2^2 &= 4 \\ x_1 + x_2 &= 2. \end{aligned} \qquad (4.30)$$

На рисунке 4.11 приведен график системы уравнений (4.30).
У данной системы существует два решения: $\overline{x}^{(1)} = \{2, 0\}$ и $\overline{x}^{(2)} = \{0, 2\}$.

1. Следуя приведенному алгоритму зададим начальное приближение $\overline{x}^{(0)} = \{1, 4\}$ и точность $\varepsilon = 0.0001$.
2. Из (4.28) найдем $\triangle x_i^{(0)}$. Вычислим якобиан $J(x)$:

$$J(x) = \begin{pmatrix} 2x_1 & 2x_2 \\ 1 & 1 \end{pmatrix}.$$

Система (4.28) имеет вид: $J(x^{(0)}) \cdot \triangle x^{(0)} = -F(x^{(0)})$, или

$$\begin{pmatrix} 2 & 8 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \triangle x_1^{(0)} \\ \triangle x_2^{(0)} \end{pmatrix} = -\begin{pmatrix} 13 \\ 3 \end{pmatrix}.$$

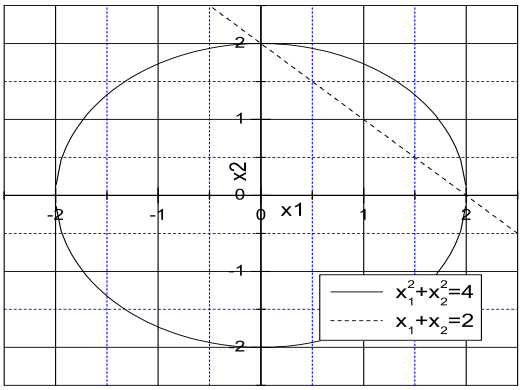

Рис. 4.11:

С помощью метода Гаусса найдем решение этой системы линейных уравнений.

$$\triangle x^{(0)} = \begin{pmatrix} \triangle x_1^{(0)} \\ \triangle x_2^{(0)} \end{pmatrix} = \begin{pmatrix} -1.833333 \\ -1.166667 \end{pmatrix}.$$

3. Найдем следующее приближение $x^{(1)}$.

$$x^{(1)} = x^{(0)} + \triangle x^{(0)} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} + \begin{pmatrix} -1.833333 \\ -1.166667 \end{pmatrix} = \begin{pmatrix} -0.8333335 \\ 2.833333 \end{pmatrix}$$

4. Так как $\delta = \max\{|-1.833333|, |-1.166667|\} = 1.833333 > \varepsilon$, переходим к пункту 2.

5. Из (4.28) вычислим $J(x^{(1)}) \cdot \triangle x^{(1)} = -F(x^{(1)})$.

$$\begin{pmatrix} -1.666667 & 5.666667 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \triangle x_1^{(1)} \\ \triangle x_2^{(1)} \end{pmatrix} = - \begin{pmatrix} 4.722223 \\ 0.0 \end{pmatrix}.$$

6. Найдем новое значение $\triangle x^{(1)}$:

$$\triangle x^{(1)} = \begin{pmatrix} 0.6439395 \\ -0.6439395 \end{pmatrix}.$$

7. Новое приближение $x^{(2)}$ равно:

$$x^{(2)} = \begin{pmatrix} -0.1893940 \\ 2.189394 \end{pmatrix}.$$

Результаты дальнейших вычислений, проведенных по программе представлены ниже.

```
 F(i)
  -4.722223      0.0000000E+00
k = 0
 DeltaX()
  0.6439395     -0.6439395
 X()
 -0.1893940       2.189394
k = 1
 Jacoby()
 -0.3787880       4.378788
   1.000000       1.000000

 F(i)
 -0.8293161      0.0000000E+00
k = 2
 DeltaX()
  0.1743148     -0.1743149
 X()
 -1.5079185E-02   2.015079
k = 3
 Jacoby()
 -3.0158371E-02   4.030158
   1.000000       1.000000

 F(i)
 -6.0770843E-02  1.6391277E-07
k = 4
 DeltaX()
  1.4967187E-02 -1.4967021E-02
 X()
 -1.1199806E-04   2.000112
k = 6
```

```
 Jacoby()
 -2.2399612E-04   4.000224
   1.000000       1.000000

 F(i)
 -4.4825202E-04 -5.8673322E-08
k = 6
 DeltaX()
  1.1186978E-04 -1.1205046E-04
 X()
 -1.2827513E-07   2.000000
 k = 6  x(k)
 -1.2827513E-07   2.000000
```

Программа 4.6

```
    program NewtonSystem
      integer,parameter::n=2
      real(4):: a(n,n),b(n),x(n)=0.0, x0(n)=0
      real(4):: Jacoby(n,n), F(n), deltaX(n), delta, eps = 0.0001
      x0(1) = 1.0 ; x0(2) = 4.0
      lend = 10
      open(7,file='rezult.dat')
     do k = 1,lend
!  Вычисление матрицы Якоби
       m = 0
       do i = 1,n
         do j = 1,n
           m = m+1
           Jacoby(i,j) = g1(m,n,x0)
         enddo
       enddo
       do i = 1,n
         F(i) = -g2(i,n,x0)
       enddo
       call Gauss(n,Jacoby,F,deltaX)
       delta = maxval( abs(deltaX))
      if( delta <= eps ) goto 10
       x0 = x0 + deltaX
       write(7,*)
```

```
       write(7,*) 'Jacoby'
       do i = 1,n
         write(7,*) (Jacoby(i,j),j=1,n)
       enddo
        write(7,*)
        write(7,*) 'F(i)'
        write(7,*) (F(j),j=1,n)
        write(7,*)
        write(7,*) 'DeltaX'
        write(7,*) (deltaX(j),j=1,n)
        write(7,*) 'X'
        write(7,*) (x0(j),j=1,n)
      enddo
       write(7,*)
10     write(7,*) 'k = ',k,' x(k) '
       write(7,*) (x0(j),j=1,n)
    end program NewtonSystem

    subroutine Gauss(n,a,b,x)
!  Решение  системы линейных уравнений методом Гаусса
      real(4)::a(n,n), a1(n,n), b(n), b1(n), p(n),x(n)
       a1 = a
       b1 = b
       do k = 1,n-1
         p(k+1:n)=a1(k+1:n,k)/a1(k,k)
        do i = k+1,n
         a1(i,k+1:n)=a1(i,k+1:n)-a1(k,k+1:n)*p(i)
         b1(i) = b1(i)-b1(k)*p(i)
        enddo
       enddo
       x(n) = b1(n)/a1(n,n)
      do k=n-1,1,-1
        s=sum(a1(k,k+1:n)*x(k+1:n))
        x(k) = (b1(k)-s)/a1(k,k)
      enddo
    end subroutine Gauss

 function g1(k,n,x)
! Вычисление матрицы Якоби
      integer::k,n
```

```
      real(4)::x(n)
     select case (k)
      case (1)
       g1 = 2.0*x(1)
      case (2)
       g1 = 2.0*x(2)
      case (3)
       g1 = 1.0
       case (4)
       g1 = 1.0
      end select
    end function g1

    function g2(k,n,x)
!  Вычисление значений функций
      real::x(n)
      select case (k)
       case(1)
        g2 = x(1)*x(1)+x(2)*x(2)-4.0
       case (2)
        g2 = x(1)+x(2)-2.0
      end select
    end function g2
```

Решение системы нелинейных уравнений методом Ньютона зависит от выбора начального приближения. Так при выборе начального значения $\overline{x}^{(0)} = \{2, 2\}$ решение расходится.

Метод Ньютона для решения систем нелинейных уравнений можно модифицировать для упрощения расчетов. При этом скорость сходимости уменьшается. Модификация метода Ньютона для решения системы нелинейных уравнений заключается в том, что матрица Якоби вычисляется один раз, с начальным приближением $x^{(0)}$. Алгоритм решения остается таким же, как и в приведенном выше методе Ньютона.

4.4. Упражнения

1. Найти интервал отделимости корней и вещественные корни уравнений:

   1. $x^4 - 3x^2 + 75x - 10000 = 0$.

2. $e^{2x} - 2e^x + 1 = 0$.

3. $x^3 - 3x - 2 = 0$.

2. Найти вещественные корни уравнений:

1. $f(x) = \sin(x) - 2\cos(x)$ на интервале [-2,2].

2. $f(x) = x - \cos(x)$ на интервале [-2,2].

3. $f(x) = (x-2)^2 - \ln(x)$ на интервале [0.5,4.5].

4. $f(x) = x^2 - e^x$ на интервале [-2,2].

3. Графически найти интервал отделимости корней и вещественные корни уравнений:

1. $f(x) = 1000000x^3 - 111000x^2 + 1110x - 1$.

2. $f(x) = 5x^{10} - 38x^9 + 21x^8 - 5\pi x^6 - 3\pi x^5 - 5x^2 + 8x - 3$.

4. Методом простой итерации найти вещественные корни уравнения:

$$x^3 - x^2 - 9x + 9 = 0$$

на отрезках: $[-4;-2]$, $[0;2]$, $[2;4]$.

5. Методом простых итераций и итераций Зейделя найти вещественные корни системы нелинейных уравнений:

1. $$\begin{aligned} \sin(x_1 + 1) - x_2^2 &= 1, \\ 2 \cdot x_1 + \cos(x_2) &= 2. \end{aligned}$$

2. $$\begin{aligned} \mathrm{tg}(x_1 \cdot x_2 + 0.2) &= x_1^2, \\ x_1 + 2x_2 &= 1. \end{aligned}$$

3. $$\begin{aligned} \cos(x_1 + 0.5) - x_2 &= 2, \\ -2 \cdot x_1 + \sin(x_2) &= 1. \end{aligned}$$

6. Переделать программу 4.5 для решения систем нелинейных уравнений модифицированным методом Ньютона.

7. Решить следующие системы нелинейных уравнений приведенными в данной главе методами.

1. $$\begin{aligned} x_1^2 + x_2^2 &= 2, \\ e^{x_1 - 1} + x_2^3 &= 2. \end{aligned}$$

2.
$$\begin{aligned} 2x_1^2 - x_1x_2 - 5x_1 &= -1, \\ x_1 + 3 \cdot \lg(x_1) - x_2^2 &= 0. \end{aligned}$$

3.
$$\begin{aligned} x_1^2 + x_2^2 + x_3^2 &= 1, \\ 2x_1^2 + x_2^2 - 4x_3^2 &= 0, \\ 3x_1^2 - 4x_2 + x_3^2 &= 0. \end{aligned}$$

4.
$$\begin{aligned} \sin(x_1 + 1) - x_2 &= 1, \\ 2x_1 + \cos(x_2) &= 2. \end{aligned}$$

5.
$$\begin{aligned} 2x_1 + tg(x_1x_2) &= 0.0, \\ \ln(x_1) + (x_2^2 - 6)^2 &= 0.0 \end{aligned}$$

6.
$$\begin{aligned} \mathrm{tg}(x_1x_2 + 0.2) - x_1^2 &= 0, \\ x_1^2 + 2x_2^2 &= 1. \end{aligned}$$

7.
$$\begin{aligned} 2x_1^3 - x_2^2 &= 1, \\ x_1x_2^3 - x_2 &= 4. \end{aligned}$$

Глава 5

Численные методы оптимизации

5.1. Введение

Задачи оптимизации возникают в практической деятельности, когда необходимо найти минимальное или максимальное значение какой - либо величины на заданном интервале. В частности, во второй главе, мы встречались с задачей минимизации ошибок округления.

Определение 5.1 *Процесс нахождения экстремума (минимума или максимума) некоторой функции назыется оптимизацией.*

Математически это можно записать так: *Необходимо найти минимум вещественной функции* $f(x_1, x_2, \ldots, x_n)$ *на некотором множестве* S n *- мерного пространства.* Задачу поиска максимума функции $f(x)$ можно свести к задаче поиска минимума функции $g(x) = -f(x)$.

Определение 5.2 *Функцию* $f(x)$ *будем называть целевой функцией, множество* S *– допустимым множеством.*

Существует три вида оптимизационных задач:

1. Задачи безусловной отимизации.

2. Задачи условной оптимизации или задачи линейного программирования. В этом случае, кроме целевой функции $f(x)$ существуют дополнительные ограничения $g_i(x)$, $h_i(x)$, где g и h – линейны.

3. Задачи условной оптимизации нелинейного программирования. В этом случае, либо целевая функция, либо одно из ограничений нелинейны.

Задачи оптимизации связаны с решением нелинейных уравнений. Если

решается задача безусловной оптимизации, то экстремум функции достигается в точке:

$$\frac{\partial}{\partial x_i} f(x) = 0,$$

а это задача нахождения корня нелинейного уравнения.

Определение 5.3 *Будем говорить, что функция $f(x)$ имеет глобальный минимум в точке x^*, если $f(x^*) \leq f(x)$ на всем множестве x.*

Определение 5.4 *Точка x^* является точкой локального минимума, если $f(x^*) \leq f(x)$ в некоторой окрестности точки x^*, то есть $\forall |x - x^*| < \varepsilon, x \neq x^*$.*

Теорема 5.1 *Пусть функция $f(x)$ непрерывна на отрезке $[a, b]$ и дифференциируема на (a, b). Тогда*

1. *Если $f'(x) > 0$ для всех $x \in (a, b)$, то функция $f(x)$ является возрастающей на [a,b].*

2. *Если $f'(x) < 0$ для всех $x \in (a, b)$, то функция $f(x)$ является убывающей на [a,b].*

Теорема 5.2. *Пусть функция $f(x)$ определена на отрезке $[a, b]$ и имеет локальный экстремум во внутренней точке $\xi \in (a, b)$. Тогда, если $f(x)$ дифференциируема в точке $x = \xi$, то $f'(\xi) = 0$.*

Теорема 5.3. (критерий первой производной). *Пусть $f(x)$ непрерывна на отрезке $[a, b]$. Кроме того, предположим, что $f'(x)$ определена для всех $x \in (a, b)$ за исключением, возможно, точки $x = \xi$. Тогда*

1. *Если $f'(x) < 0$ на интервале (a, ξ) и $f'(x) > 0$ на (ξ, b) то $f(\xi)$ – локальный минимум.*

2. *Если $f'(x) > 0$ на интервале (a, ξ) и $f'(x) < 0$ на (ξ, b) то $f(\xi)$ – локальный максимум.*

Теорема 5.4. (критерий второй производной). *Пусть $f(x)$ непрерывна на отрезке $[a, b]$ и $f'(x)$ и $f''(x)$ определены на (a, b). Также предположим, что $\xi \in (a, b)$ – критическая точка, в которой $f'(\xi) = 0$. Тогда*

1. *Если $f''(\xi) > 0$, то значение $f(\xi)$ является локальным минимумом функции f.*

2. *Если $f''(\xi) < 0$, то значение $f(\xi)$ является локальным максимумом функции f.*

3. *Если $f''(\xi) = 0$, то этот критерий не является окончательным.*

Пример 5.1 Найдем локальные экстремумы функции $f(x) = x^3 + 6x^2 + 6x - 10$ на отрезке $[-4, 2]$. Первая производная $f'(x) = 3x^2 + 12x + 6$, вторая производная $f''(x) = 6x + 12$. Из первого уравнения следует, что существуют два корня этого уравнения, это точки $x = -0.585786$ и $x = -3.4142135$. На рисунке 5.1 приведен график функции $f(x) = x^3 + 6x^2 + 6x - 10$ на отрезке $[-4, 2]$.

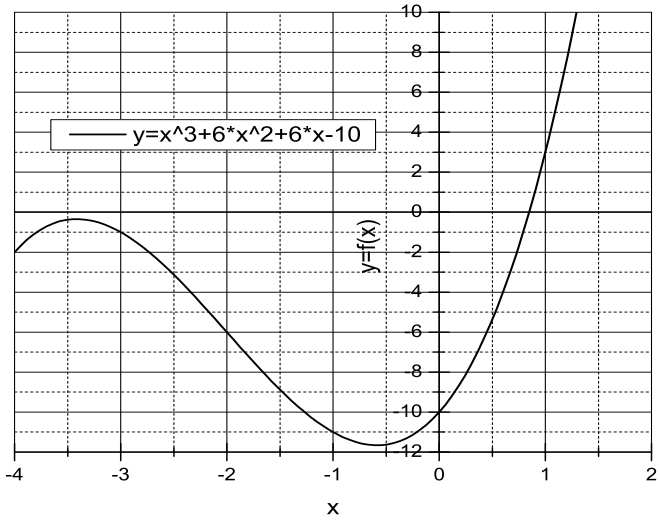

Рис. 5.1:

- **Первый корень**. В точке $x = -0.585786$ вторая производная $f''(-0.585786) = 8.485281 > 0$, отсюда следует, что функция $f(x)$ в точке $x = -0.585786$ имеет локальный минимум.

- **Второй корень**. В точке $x = -3.4142135$ вторая производная $f''(-3.4142135) = -8.485281 < 0$, отсюда следует, что функция $f(x)$ в точке $x = -3.4142135$ имеет локальный максимум.

Численные методы для решения задач оптимизации часто основаны на разложении в ряд Тейлора. Поэтому достаточно трудно отличить локальный экстремум от глобального. Это связано с тем, что разложение в ряд Тейлора дает информацию о поведении функции вблизи заданной точки, но не дает информацию о поведении функции во все области. Задачи оптимизации решаются в основном с помощью итераций и позволят находить приближенное решение. Можно показать, что правильно написанная программа завершится за конечное количество итераций.

5.2. Методы отыскания безусловного экстремума функции одного переменного.

Существует большое количество методов, позволяющих отыскать локальные экстремумы. Это метод деления отрезка пополам, метод золотого сечения, метод Ньютона и другие. Рассмотрим некоторые из этих методов, позволяющих найти локальные экстремумы функций. Первый из них – метод золотого сечения, который аналогично методу дихотомии, позволяет сократить отрезок локализации экстремума функции.

5.2.1. Метод золотого сечения

Пусть $f(x)$ – вещественная функция, заданная на отрезке $[a, b]$. Будем считать, что функция на этом отрезке унимодальна, то есть имеет на $[a, b]$ только один экстремум. Для нахождения экстремума функции методом золотого сечения не требуется дифференциируемости и непрерывности функции. Минимум функции $f(x)$ можно искать, используя метод деления отрезка пополам. Для этого необходимо разбить отрезок $[a, b]$ пополам точкой $x = (a + b)/2$, отступить от точки x вправо и влево на ε, определить направление возрастания функции и, в зависимости от этого, заменить один из концов отрезка либо a либо b на $x \pm \varepsilon$ соответственно. Однако этот метод не очень экономичен, так как вычисление значения функции в точке x в дальнейшем больше не используется.

Определение 5.5 *Функция $f(x)$ является унимодальной на отрезке $[a, b]$, если существует такое число $\xi \in [a, b]$, что*

1. *$f(x)$ убывает на $[a, \xi)$ и возрастает на $(\xi, b]$,*

2. *$f(x)$ возрастает на $[a, \xi)$ и убывает на $(\xi, b]$.*

Идея метода золотого сечения заключается в том, чтобы на каждом новом шаге использовались уже найденные значения функции $f(x)$. Без ограничения общности будем рассматривать отрезок [0,1]. Для реализации этого метода разобъем интервал $X = b - a = 1$ на две неравные части x_1 и x_2 по правилу золотого сечения. По этому правилу, отношение большего отрезка x_1 к длине интервала X должно равняться отношению меньшего отрезка x_2 к большему.

$$x_1 + x_2 = X,\ \frac{x_1}{X} = \frac{x_2}{x_1} \Rightarrow \left(\frac{x_2}{x_1}\right)^2 + \frac{x_2}{x_1} - 1 = 0. \qquad (5.1)$$

Уравнение (5.1) имеет два корня: $\frac{x_2}{x_1} = \frac{\sqrt{5}\pm 1}{2}$. Один из корней не имеет физического смысла, поэтому остается только один корень $\frac{x_2}{x_1} = \frac{\sqrt{5}-1}{2} = 0.618934$. Условие унимодальности гарантирует, что значения функции в точках $f(x_1)$ и $f(x_2)$ больше, чем минимальное значение функции на отрезке $[a, b]$.

При таком делении отрезка на каждом шаге, кроме первого, сохраняется одна точка, в которой значение функции уже вычислено на предыдущем этапе. Метод золотого сечения по скорости сходимости является таким же медленным, как и метод дихотомии. Скорости сходимости у них линейные. Но он имеет то преимущество, что он гарантированно сходится в наихудшем случае.

Программа 5.1 Метод золотого сечения.

```
   program Gold
     implicit none
! Variables
     real::x0,f,a=-1.0,b=2.0,c=1.0,eps=0.0001,xmin,golden
     EXTERNAL f
! Body of Gold
      x0 = golden(a,b,c,f,eps,xmin)
      write(*,*) 'f(x) = ',x0,'x = ',xmin
    end program Gold

    FUNCTION golden(ax,bx,cx,f,eps,xmin)
      REAL golden,ax,bx,cx,eps,xmin,f,R,C
      EXTERNAL f
      PARAMETER (R=.61803399,C=1.-R)
      REAL f1,f2,x0,x1,x2,x3
      x0=ax
```

```
      x3=cx
      if(abs(cx-bx).gt.abs(bx-ax))then
        x1=bx
        x2=bx+C*(cx-bx)
      else
        x2=bx
        x1=bx-C*(bx-ax)
      endif
      f1=f(x1)
      f2=f(x2)
1     if(abs(x3-x0).gt.eps*(abs(x1)+abs(x2)))then
        if(f2.lt.f1)then
          x0=x1
          x1=x2
          x2=R*x1+C*x3
          f1=f2
          f2=f(x2)
        else
          x3=x2
          x2=x1
          x1=R*x2+C*x0
          f2=f1
          f1=f(x1)
        endif
      goto 1
      endif
      if(f1.lt.f2)then
        golden=f1
        xmin=x1
      else
        golden=f2
        xmin=x2
      endif
      return
      end function golden

      real function f(x)
        real x
        f = 3.0*x**4-4.0*x**3 -12.0*x**2
```

```
end function f
```

Пример 5.2 Найти экстремумы функции $f(x) = 3x^4 - 4x^3 - 12x^2$ на отрезке [-2,3]. На этом отрезке функция $f(x) = 3x^4 - 4x^3 - 12x^2$ не унимодальна и имеет два минимума (рисунок 5.2) – в точках -1 и 2. В результате работы программы мы получим правый минимум $f(x) = -32$ в точке $x = 2.00001$. Точное значение минимума – $f(x) = -32$, которое достигается в точке $x = 2.0$. Для нахождения второго минимума необходимо изменить отрезок на котором ищется экстремум. Для этого зададим отрезок [-2,0]. На этом отрезке мы получим значение второго минимума $f(x) = -5.0$, который достигается в точке $x = -1.000091$.

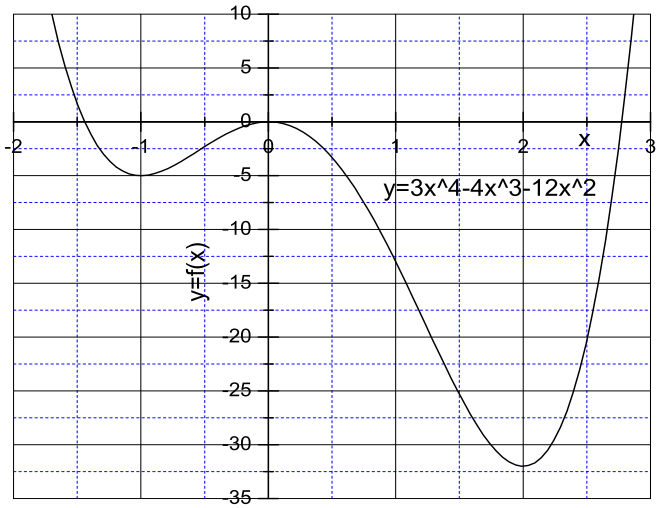

Рис. 5.2:

5.2.2. Метод Ньютона

В методах оптимизации, как и в методах отыскания корней нелинейных уравнений, существует методы, которые обеспечивают более быструю сходимость. Одним из таких методов является метод Ньютона. Для того, чтобы отыскать экстремум функции методом Ньютона, необходимо, чтобы выполнялись следующие условия:

1. Функция $f(x)$ должна быть ограничена.

2. Функция $f(x)$ должна быть дважды дифференциируема на отрезке $[a, b]$.

Основной идеей метода Ньютона является замена функции $f(x)$, для которой мы ищем минимум, более простой функцией. Линейная функция не подходит для такой замены, так как она не имеет конечного минимума. Поэтому, в качестве такой более простой функции берется квадратичная функция.

Используя разложение в ряд Тейлора для x_i приближения функции $f(x)$ можно записать:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \dots .$$

Как известно, функция имеет экстремум в тех точках, в которых первая производная обращается в нуль. Поэтому, если оставить два члена разложения функции в ряд Тейлора и продифференциировать по x получим:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}. \tag{5.2}$$

Метод Ньютона имеет квадратичную сходимость, но его недостатком является необходимость вычисления первой и второй производной функции $f(x)$.

Пример 5.3 Нахождение экстремума функции методом Ньютона. Рассмотрим функцию $f(x) = x^2 - sin(x)$ на отрезке $[0, 3]$. С помощью программы 5.2 найдем минимум этой функции. Методом Ньютона минимум функции $f(x) = x^2 - sin(x)$ на отрезке $[0, 3]$ находится за 5 итераций. Этот минимум достигается в точке $x = 0.4501837$. Значение функции в точке минимума $f(x) = -0.2324656$.

Ниже приведены результаты работы программы 5.2.

```
num=          1  x =  -0.2646432      f(x) =   0.3316009
num=          2  x =   0.5950221      f(x) =  -0.2064757
num=          3  x =   0.4536808      f(x) =  -0.2324507
num=          4  x =   0.4501859      f(x) =  -0.2324656
num=          5  x =   0.4501836      f(x) =  -0.2324656
```

На рисунке 5.3 приведен график функции $f(x) = x^2 - sin(x)$ на отрезке $[0, 4]$.

Программа 5.2 Метод Ньютона.

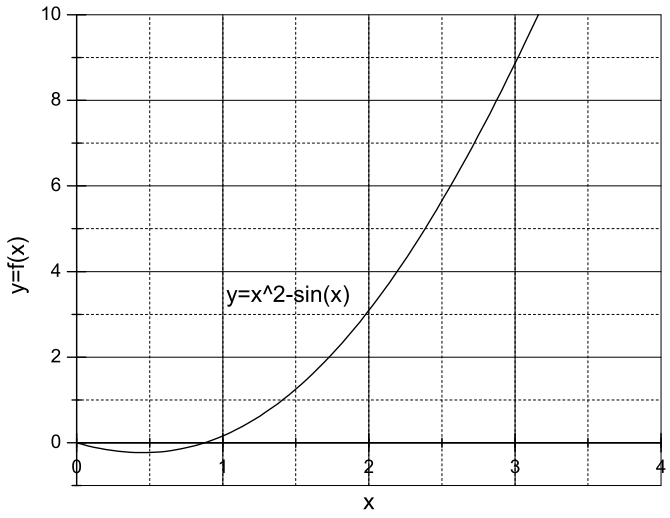

Рис. 5.3:

```
  program Newton

!  Ищем минимум функции f(x)=x*x-sin(x) на отрезке [0,3]

    ! Variables
      real::x0,f,f1,f2,fx1,fx2,eps=1.0e-06
      integer::num=0,imax=50
      open(7,file='Newton.dat')
    ! Body of Newton
      x0=3.0
     do
       fx1=f1(x0)
       fx2=f2(x0)
       num=num+1
      if( fx2 == 0.0 ) then
        write(*,*) "Secondary derivative is equal to zero"
        stop
      elseif( num > imax ) then
        write(*,*) "You exeed max iterations"
        stop
```

```
  endif
  if( abs(fx1) < eps ) exit
   x0 = x0-fx1/fx2
   write(7,*) ' num=',num,' x = ',x0,' f(x) = ',f(x0)
   enddo
end program Newton

real function f(x)
  f = x*x-sin(x)
end

real function f1(x)
  f1 = 2*x-cos(x)
end

real function f2(x)
  f2 = 2.0 + sin(x)
end
```

Несмотря на то, что для применения метода Ньютона необходимо вычислить первую и вторую производную, быстрая сходимость этого метода делает его привлекательным. Несложными модификациями его можно превратить в эффективный метод, который на каждой итерации обеспечивает продвижение к решению. В **Главе 4** приведена модификация метода Ньютона для решения нелинейных уравнений. Проведем такую же модификацию и для задачи нахождения экстремума функции. Для этого, вместо $f'(x)$ воспользуемся правосторонней разностью:

$$f'(x_i) \approx \frac{f(x_i + h) - f(x_i)}{h}. \tag{5.3}$$

Величину шага h удобно принять равной $h \approx \sqrt{\varepsilon_m}$.

Для повышения точности, вместо (5.3) лучше использовать формулу центральной разности:

$$f'(x_i) \approx \frac{f(x_i + h) - f(x_i - h)}{2h}. \tag{5.4}$$

Аппроксимация $f'(x)$ центральной разностью дает возможность довести порядок точности до $O(h^2)$, вместо $O(h)$ для уравнения (5.3). В качестве h в этом случае можно выбрать значение $\sqrt[3]{\varepsilon_m}$. Для второй производной

можно воспользоваться аппроксимацией:

$$f''(x_i) \approx \frac{f(x_i+h) - 2f(x_i) + f(x_i-h)}{h^2}.$$

Переделаем **Программу 5.2** с учетом сделанных выше предложений.

Программа 5.3 Модифицированный метод Ньютона.

```
program Newton2
  implicit none
! Variables
  real:: x0,fx1,fx2,h,h1,eps=1.0e-06,f1
  integer:: num, imax=25
! Body of Newton2
  x0 = 0.25
  h=sqrt(eps)
  h1 = eps**(1./3.)
 do
   fx1 = (f1(x0+h)-f1(x0-h))/(2.0*h)
   fx2 = (f1(x0+h1)-2.0*f1(x0)+f1(x0-h1))/(h1*h1)
  if( abs(fx1) < eps ) then
    exit
  else
    num = num + 1
    x0 = x0-fx1/fx2
  endif
  if( num > imax ) then
    write(*,*) "You exeed max iterations = ",num
    stop
  endif
 enddo
 write(*,*) ' num = ',num,' x = ',x0, ' f(x) = ',f1(x0)
end program Newton2

real function f1(x)
  f1 = 1.0/((x-0.3)**2+0.01)+1.0/((x-0.9)**2+0.04)-6.0
end
```

Пример 5.4 Найти локальный минимум функции

$$f(x) = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6, \qquad (5.5)$$

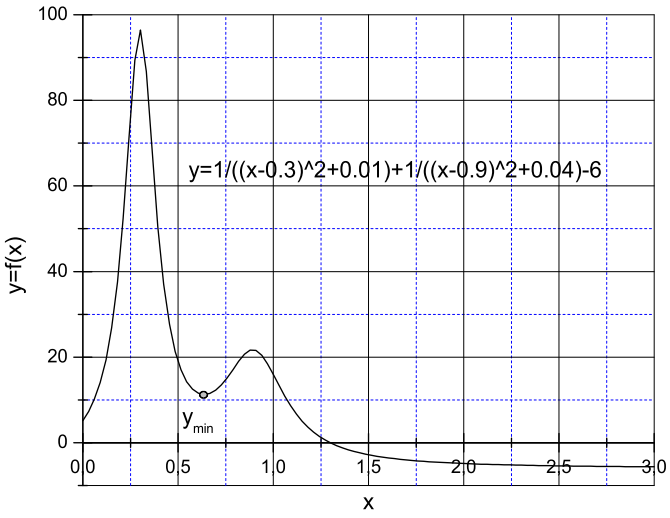

Рис. 5.4:

на отрезке $[0.25, 1]$.

На рисунке 5.4 приведен график функции (5.5) на отрезке [0,3]. На рисунке отмечен локальный минимум данного уравнения на указанном отрезке.

Приведем рассчитанную по программе 5.3 таблицу значений аргумента x и соответствующие значение функции

num	x	f(x)
1	0.5723462	19.38345
2	0.4506198	12.66662
3	0.6248187	11.29867
4	0.6366917	11.25279
5	0.6370096	11.25276

Модифицированным методом Ньютона мы получаем значение максимума функции за пять итераций.

Если воспользоваться аппроксимациями (5.3), (5.2), то полученные оценки будут менее точными. В том случае, когда мы находимся далеко от решения, где $f'(x) \neq 0$ замена функций ее разностными аппроксимациями не оказывает существенного влияния на точность. Однако вблизи экстремума точность может существенно снизиться, так как $f'(x) \approx 0$.

5.3. Численные методы отыскания экстремума функции нескольких переменных.

Определение 5.4 можно обобщить на случай функции нескольких переменных:

Определение 5.6 *Пусть функция $f(x,y)$ определена в области*

$$R = \{(x,y) : (x-\xi)^2 + (y-\eta)^2 < r^2\}.$$

Функция $f(x,y)$ имеет локальный минимум в точке (ξ,η), если

$$f(\xi,\eta) \le f(x,y), \quad \textit{для каждой точки } (x,y) \in R.$$

Функция $f(x,y)$ имеет локальный максимум в точке (ξ,η), если

$$f(\xi,\eta) \ge f(x,y), \quad \textit{для каждой точки } (x,y) \in R.$$

Теорема 5.5. (критерий второй производной). *Предположим, что функция $f(x,y)$ имеет первую и вторую непрерывные частные производные в области R. Пусть $f(\xi,\eta)$ – критическая точка в которой $f_x(\xi,\eta) = 0$ и $f_y(\xi,\eta) = 0$. Тогда*

1. *Если $f_{xx}(\xi,\eta)\cdot f_{yy}(\xi,\eta) - f^2_{xy}(\xi,\eta) > 0$, и $f_{xx} > 0$, то $f(\xi,\eta)$ локальный минимум функции f.*

2. *Если $f_{xx}(\xi,\eta)\cdot f_{yy}(\xi,\eta) - f^2_{xy}(\xi,\eta) > 0$, и $f_{xx} < 0$, то $f(\xi,\eta)$ локальный максимум функции f.*

3. *Если $f_{xx}(\xi,\eta)\cdot f_{yy}(\xi,\eta) - f^2_{xy}(\xi,\eta) < 0$, то функции $f(\xi,\eta)$ не имеет локального экстремума в точке ξ,η*

4. *Если $f_{xx}(\xi,\eta)\cdot f_{yy}(\xi,\eta) - f^2_{xy}(\xi,\eta) = 0$, то этот критерий не является окончательным.*

Один из методов отыскания экстремума функции нескольких переменных заключается, как и в одномерном случае, в решении системы нелинейных уравнений относительно частных производных этой функции:

$$\frac{\partial f}{\partial x_i} = 0, \ i = 1, 2, \dots, n. \tag{5.6}$$

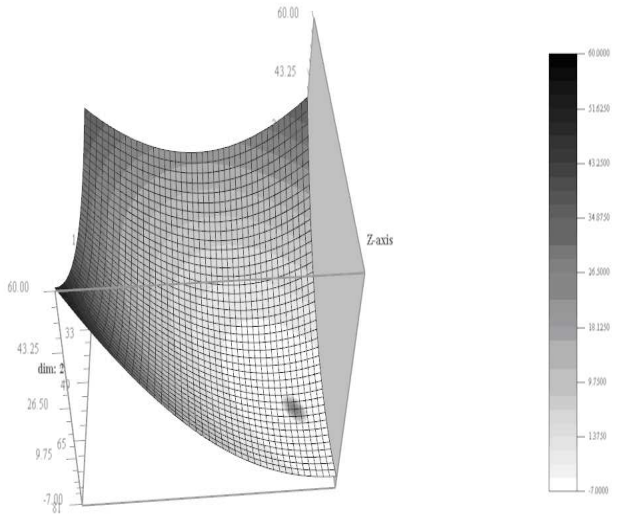

Рис. 5.5:

Пример 5.5 Найдем минимум функции двух переменных

$$f(x,y) = x^2 - 4x + y^2 - y - xy. \tag{5.7}$$

Частные производные этой функции равны:

$$\frac{\partial f}{\partial x} = 2x - 4 - y$$

и

$$\frac{\partial f}{\partial y} = 2y - 1 - x.$$

Приравняв частные производные нулю получим систему уравнений:

$$\begin{aligned} 2x - y &= 4 \\ -x + 2y &= 1. \end{aligned}$$

Решением этой системы будут значения: $x = 3,\ y = 2$. Найдем вторые производные:

$$\begin{aligned} f_{xx} &= 2, \\ f_{xy} &= -1, \\ f_{yy} &= 2. \end{aligned}$$

Это первый случай **Теоремы 5.5**, то есть

$$f_{xx}(3,2)\cdot f_{yy}(3,2) - f_{xy}^2(3,2) = 3 > 0 \text{ и } f_{xx}(3,2) = 2 > 0.$$

Отсюда следует, что $f(x,y)$ имеет локальный минимум, равный -7 в точке (3,2). На рисунке 5.5 представлен график функции (5.7) двух переменных $f(x,y)$.

Условие (5.6) является необходимым условием экстремума функции $f(x)$. Однако методы решения системы нелинейных уравнений часто оказываются медленно сходящимися.

Поэтому более часто используются другие методы поиска минимума функции. Как и в одномерном случае поиск максимума функции $f(x) = 0$ можно свести к поиску минимума, решая уравнение относительно $g(x) = -f(x)$. В двумерном случае функция $f(x,y) = 0$ представляет собой поверхность с "холмами"и "оврагами". Поиск минимума функции подобен спуску в овраг, поэтому такие методы называются методами спуска.

5.3.1. Метод покоординатного спуска

Одним из наиболее простых методов поиска экстремума функции нескольких переменных, является метод покоординатного спуска. Идея этого метода заключается в том, чтобы последовательно фиксировать $n-1$ координату и осуществлять поиск минимума по одной координате. В таком случае можно применить один из известных методов нахождения минимума функции одного переменного. Продвинувшись на один шаг по одной координате x_i, выбираем направление движения по другой координате. Продолжаем двигаться последовательно по всем координатам до тех пор, пока не пройдем все координатные направления. И так повторяется до тех пор, пока мы не придем в точку экстремума. Существуют разновидности этого метода:

1. движение осуществляется с постоянными шагами;
2. движение осуществляется с переменными шагами, как по координатам, так и по итерациям.

Математически это можно записать так. Вычисляем производную функции $f(x)$ по координате x_1 и находим первое приближение:

$$x^{(1)} = x^{(0)} - \alpha_1 \frac{\partial f}{\partial x_1}\Big|_{x=x^{(0)}} \cdot \overline{e}_1, \quad \text{где } \overline{e}_1 - \text{единичный вектор по оси } x_1. \tag{5.8}$$

На втором шаге фиксируем все координаты, кроме x_2 и вычисляем:

$$x^{(2)} = x^{(1)} - \alpha_2 \frac{\partial f}{\partial x_2}\Big|_{x=x^{(1)}} \cdot \overline{e_2}.$$

Повторяем вычисления по всем координатам. Здесь α_i шаг по i - ой координате. Этот шаг выбирается из условия сходимости последовательности (5.8):

$$f(x^{(i+1)}) > f(x^{(i)}) - \alpha_i \frac{\partial f}{\partial x^{(i)}}. \tag{5.9}$$

Выбор шага α_i оказывает большое влияние на процесс нахождения минимума функции. Если шаг большой – может не выполнится условие (5.9), либо могут возникнуть колебания вокруг точки минимума. Если шаг взять очень маленьким – это приведет к неоправданно большому количеству итераций. Построим итерационный процесс, который позволяет выбрать последовательность α_i. Выбираем начальный шаг α_0. С этим шагом начинаем проводить итерации и для каждой итерации проверяем условие (5.9). Если условие (5.9) выполняется, продолжаем проводить итерации с этим шагом. Если условие (5.9) нарушается, шаг α_0 уменьшаем до тех пор, пока не выполнится условие (5.9). На рисунке 5.6 представлен график функции (5.10)

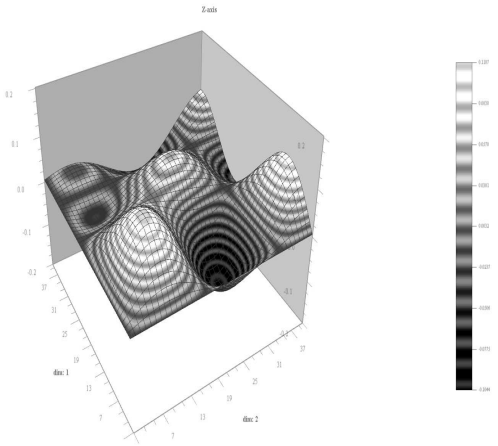

Рис. 5.6:

Таким образом, в этом методе на каждом шаге движение к экстремуму осуществляется только по одной координате. Геометрически, для случая n=2 этот метод можно представить так (рисунок 5.7). Из начальной

точки $\{X1_0, X2_0\}$ движемся параллельно координате X2, находим точку $\{X1_1, X2_1\}$ локального минимума вдоль этого направления. Фиксируем эту точку и начинаем двигаться вдоль оси X1 до точки минимума в этом направлении. Такими последовательными шагами мы движемся к точке минимума функции (5.10).

Пример 5.6 Методом покоординатного спуска найти локальный минимум функции изображенной на рисунке 5.6:

$$f(x, y) = \exp(x^2 + y^2) + 2x - 3.5y. \tag{5.10}$$

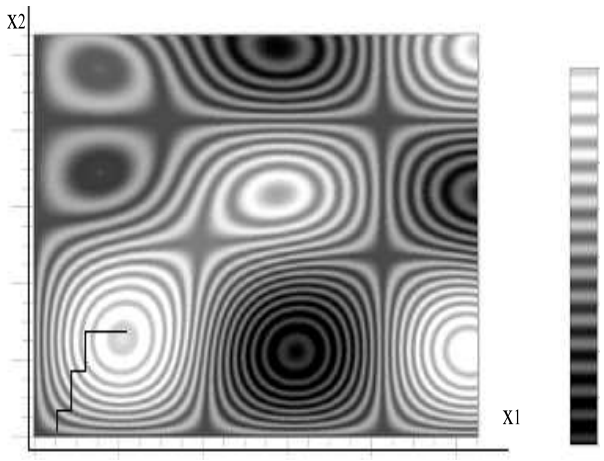

Рис. 5.7:

Локальный минимум этой функции находится в точке с координатами $x = -0.4456204$ и $y = 0.7804637$. Значение функции в этой точке $f(x, y) = -1.380118$. Ниже приведен текст программы для поиска минимума функции нескольких переменных методом покоординатного спуска. В модуле приводятся описания глобальных переменных. В главной программе задаются начальные данные и происходит обращение к подпрограмме coord, которая осуществляет поиск минимума по одной оси. Вторая координата зафискирована как параметр. Поиск минимума осуществляется с помощью подпрограммы Gold1. Эта подпрограмма является модификацией программы 5.1.

Программа 5.4 Метод покоординатного спуска.

```
module one
 integer,parameter:: nx=10
 integer i,mx
```

```
     real::z(nx),d,k
    end module one

    Program Coord1
      use one
      external f
      real::eps = 1.0E-5, delta = 1.0E-5
      real,allocatable::z0(:),z1(:),z2(:)
      mx = 2
      allocate( z0(mx), z1(mx), z2(mx) )
      z0(1) = -1.0
      z1(1) = 0.0
      z2(1) = 0.0
      z0(2) = 0.0
      z1(2) = 1.0
      z2(2) = 0.0
      call coord(eps,delta,z0,z1,z2,f)
     do i = 1,mx
        write(*,*) i,z2(i)
     enddo
      write(*,*) d
    end Program Coord1

    subroutine coord(eps,delta,z0,z1,z2,f)
      use one
      real::z0(mx),z1(mx),z2(mx)
      do i = 2,mx
        z(i) = z2(i)
      enddo
12     l = 0
       do i = 1,mx
         call GOLD1(z0(i),z1(i),eps,x,f)
        if( abs(x-z2(i)) > eps ) l=1
          z2(i) = x
       enddo
        if( l == 1 ) goto 12
    end subroutine coord

    function f(x)
```

```
      use one
      z(i) = x
      f = exp(z(1)**2+z(2)**2)+2.*z(1)-3.5*z(2)
      d = f
    end

    subroutine GOLD1(a,b,eps,x,f)
!  Подпрограмма поиска минимума одномерной функции методом
золотого сечения
      g = (sqrt(5.0)-1)/2.0    !0.618034
      r = (b-a)*g
      x1 = a+r
      f1 = f(x1)
      x2 = b-r
      f2 = f(x2)
      r = r*g
     do while( r > eps )
     if( f1 <= f2 ) then
       x = x2+r
       x2 = x1
       f2 = f1
       f1 = f(x)
       x1 = x
      else
       x = x1-r
       x1 = x2
       f1 = f2
       f2 = f(x)
       x2 = x
      endif
      r = r*g
      enddo

    end subroutine GOLD1
```

5.3.2. Градиентные методы

В градиентных методах, вместо поочередного движения по координатам, используют дополнительную информацию о том направлении, в котором

функция убывает быстрее всего. Такую информацию дает градиент, построенный в очередной точке. Градиент функции $f(x)$ в точке $x^{(k)}$ равен

$$grad \cdot f(x^{(k)}) = \frac{\partial f(x^{(k)})}{\partial x^{(k)}}. \tag{5.11}$$

Для поиска минимума функции, движение на очередном шаге задается в направлении, противоположном градиенту. Его можно вычислить по формуле, аналогичной (5.8):

$$x^{(k+1)} = x^{(k)} - \alpha_k \frac{\partial f(x^{(k)})}{\partial x^{(k)}}. \tag{5.12}$$

Определение 5.7 *Методы, в которых направление движения на каждом шаге определяется в направлении, который задается градиентом, называются градиентными.*

Как и в методе покоординатного спуска очень важным фактором, влияющим на сходимость, является выбор величины шага. Выбор шага можно осуществить с помощью процедуры, описанной в предыдущем параграфе. Основные отличия от метода координатного спуска заключаются в том, что движение идет не по координате, а по направлению наискорейшего убывания функции.

5.3.3. Метод наискорейшего спуска

Градиентный метод (5.12) можно ускорить, если на каждой итерации выбирать шаг из условия минимума функции $f(x)$ вдоль антиградиента. Таким образом, из (5.12) можно получить:

$$f(x(k) - \alpha_k \frac{\partial f(x^{(k)})}{\partial x^{(k)}}) = \min(f(x(k) - \alpha_k \frac{\partial f(x^{(k)})}{\partial x^{(k)}})) \tag{5.13}$$

Так как на каждом на каждом шаге приходится решать задачу минимизации функции (5.12), то метод наискорейшего спуска требует больших вычислительных затрат. Кроме того, программирование такого алгоритма усложняется. Однако, в целом, этот метод позволяет быстрее найти минимум функции, так как на каждом шаге мы выбираем оптимальную стратегию движения.

Пример 5.7 Возьмем функцию $f(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2$. Минимум этой функции достигается в точке $x^* = \{0.0, 0.0, 0.0\}$. Вычислим градиент функции:

$$\nabla(\overline{x}) = grad \cdot f(\overline{x}) = \{2x_1, 2x_2, 2x_3\}.$$

На каждом шаге нам необходимо решать задачу одномерной оптимизации (5.13):

$$\min_{\alpha} f(\overline{x} - \alpha \nabla f(\overline{x})) = (x_1 - 2\alpha x_1)^2 + (x_2 - 2\alpha x_2)^2 + (x_3 - 2\alpha x_3)^2.$$

Минимум этой функции можно вычислить, если взять от нее производную и положить ее равной нулю:

$$\alpha = \frac{x_1^2 + x_2^2 + x_3^2}{2(x_1^2 + x_2^2 + x_3^2)} = \frac{1}{2}.$$

Если в качестве начальной точки взять точку $\{2.0, 2.0, 2.0\}$, то мы сразу получим точку минимума:

$$(2.0 - \frac{1}{2} \cdot 2.0 \cdot 2.0) - ((0)^2 + (0)^2 + (0)^2) = 0.$$

Ситуация значительно осложняется, если функция не такая хорошая.

Пример 5.8 Возьмем функцию

$$f(x_1, x_2, x_3) = x_1^2 + 10x_2^2 + 100x_3^2.$$

Минимум этой функции находится в точке $\overline{x} = \{0.0, 0.0, 0.0\}$. Проведем такие же вычисления, как и в примере 5.7.

$$\nabla f(\overline{x}) = \{2x_1, 20x_2, 200x_3\}.$$

Теперь задачу оптимизации необходимо провести для:

$$\min_{\alpha} f(\overline{x} - \alpha \nabla \cdot f(\overline{x})) = (x_1 - 2\alpha x_1)^2 + (x_2 - 20\alpha x_2)^2 + (200\alpha x_3)^2$$

Отсюда

$$\alpha = \frac{x_1^2 + 10^3 x_2^2 + 10^4 x_3^2}{2(x_1^2 + 10^3 x_2^2 + 10^4 x_3^2)}.$$

В этом случае найти явную формулу для α не удается. Несколько первых итераций приведены в таблице:

итерация	x	f(x)
0	{1.0000,1.0000,1.0000}	111.0000
1	{0.9899,0.8991,-0.0091}	9.0718
2	{0.8981,0.0661,0.0751}	1.4148
3	{0.8890,0.0593,-0.0016}	0.8258
4	{0.7368,-0.0422,0.0255}	0.6260

В этом случае метод сходится достаточно медленно. Для получения приемлимого результата $\{10^{-5}, 10^{-6}, 10^{-3}\}$ необходимо около 200 итераций. Таким образом, для функций, имеющих овражный характер, градиентные методы сходятся плохо [1].

Во всех рассмотренных методах сходимость к локальному минимуму $\overline{x}^*$ существует для любой начальной точки. Если функция $f(\overline{x})$ ограничена снизу, то ее градиент удовлетворяет условию Липшица:

$$\|f'(x) - f'(y)\| \leqslant M\|\overline{x} - \overline{y}\|, \tag{5.14}$$

для любых $\overline{x}, \overline{y} \in$ допустимому множеству, M – константа.

Ускорить сходимость можно, используя n - мерные аналоги метода Ньютона. Как и для одномерного случая этот метод основан на квадратичной аппроксимации функции $f(\overline{x})$. Эта аппроксимация получена разложением в ряд Тейлора вблизи точки $\overline{x}_0$. Приведем для примера, разложение функции двух переменных:

$$f(x_1 + h_1, x_2 + h_2) \approx f(x_1, x_2) + h_1\frac{\partial f(x_1, x_2)}{\partial x_1} + h_2\frac{\partial f(x_1, x_2)}{\partial x_2} +$$
$$\frac{1}{2}\Big(h_1^2\frac{\partial^2 f(x_1, x_2)}{\partial x_1^2} + h_1h_2\frac{\partial^2 f(x_1, x_2)}{\partial x_1 \partial x_2} + h_2h_1\frac{\partial^2 f(x_1, x_2)}{\partial x_2 \partial x_1} + h_2^2\frac{\partial^2 f(x_1x_2)}{\partial^2 x_2^2}\Big) + \dots .$$

Обозначим через $\nabla^2 f$ – матрицу вторых производных в точке x^* – матрицу Гессе:

$$\Big(\nabla^2 f\Big)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

$$f(\overline{x} + h) \approx f(\overline{x}) + h\nabla f(\overline{x}) + \frac{1}{2}h^2\nabla^2 f(\overline{x}). \tag{5.15}$$

Формула (5.15) верна не только в двумерном случае, но и в многомерном. Таким образом, метод Ньютона в многомерном случае выглядит также, как и в одномерном:

$$\overline{x}_{k+1} = \overline{x}_k - \frac{\nabla f(\overline{x}_k)}{\nabla^2 f(\overline{x}_k)}. \tag{5.16}$$

Здесь $\nabla^2 f(\overline{x})$ – является гессианом функции $f(\overline{x})$.

В отличии от градиентных методов, метод Ньютона имеет квадратичную сходимость:

$$\|x_{k+1} - x^*\| \leqslant \beta\|x_k - x^*\|^2,$$

где $\beta > 0$ – константа, зависящая от $f(\overline{x})$.

Сходимость метода Ньютона в многомерном случае подтверждается следующей теоремой:

Теорема 5.6. *Пусть x^* решение системы нелинейных уравнений $f(\overline{x}) = 0$ такое, что якобиан функции невырожден, а вторые частные производные функции $f(\overline{x})$ – непрерывны в окрестности точки x^*. Тогда, если x^0 достаточно близко к x^*, то итерации метода Ньютона сходятся.*

Окончание процесса вычислений можно определить при выполнении хотя бы одного из условий:

$$\|\overline{x}_{k+1} - \overline{x}_k\| \leqslant \varepsilon_1, \quad \|\frac{\partial f}{\partial \overline{x}}\Big|_{\overline{x}=\overline{x}_{(k)}}\| \leqslant \varepsilon_2, \quad \|f(\overline{x}_{k+1}) - f(\overline{x}_k)\| \leqslant \varepsilon_3.$$

Где $\varepsilon_1,\ \varepsilon_2,\ \varepsilon_3$ – точность для каждого условия.

Пример 5.9 Найти минимум функции:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{5.17}$$

Начальное значение $x_0 = (-2.0, 5.0)$. Якобиан этой функции имеет вид:

$$\nabla f(\overline{x}) = \begin{pmatrix} -400x_1(x_2 - x_1)^2 + 2(x_1 - 1) \\ 200(x_2 - x_1^2) \end{pmatrix}.$$

Гессиан имеет вид:

$$\nabla^2 f(\overline{x}) = \begin{pmatrix} 1200x_1(x_1^2 - 400x_2) + 2 & -400x_1 \\ -400x_1 & 200 \end{pmatrix}.$$

Применяя метод Ньютона (5.16) к функции (5.17), найдем, что минимум функции достигается в точке $\{1.0, 1.0\}$, и равен $f(x_1, x_2) = 0.0$.

Метод Ньютона имеет второй порядок сходимости, в тех случаях, когда этот метод сходится. Недостатком метода является необходимость вычисления первых производных и матрицу вторых производных. На практике используют комбинацию градиентных методов и метода Ньютона.

Алгоритм поиска многомерной минимума функции. Пусть x_0 – начальное приближение.

1. Находим значение функции в точке x_k и ее градиент $f_k = f_k(x_k), \quad \nabla f_k = \nabla f(x_k)$ и, если минимум найден - заканчиваем.

2. Находим направление q: такое, что $f(x_k + \varepsilon q) < f_k$, для малых ε.

3. Находим такое $\alpha > 0$, что $f(x_k + \alpha q) < f_k$: $x_{k+1} = x_k + \alpha q, \quad k = k+1$, и переходим к 1.

5.4. Упражнения

1. Используя теоремы 5.3 и 5.4 найти локальные экстремумы для следующих функций на заданном интервале.

1. $f(x) = e^x/x^2$; $[0.5, 3]$.
2. $f(x) = x + 3/x^2$; $[0.5, 3]$.
3. $f(x) = 4x^3 - 8x^2 - 11x + 5$; $[0, 2]$.
4. $f(x) = (x + 2.5)/(4 - x^2)$; $[-2.0, 2.0]$.
5. $f(x) = -2\sin(x) + \sin(2x) - 2\sin(3x)/3$; $[1; 3]$.

2. Используя теорему 5.5 найти локальные минимумы для следующих функций:

1. $f(x, y) = x^3 + y^3 - 3x - 3y + 5$.
2. $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$.
3. $f(x, y) = x^2 + y^2 + x - 2y - xy + 1$.
4. $f(x, y) = x^2y = xy^2 - 3xy$.

3. Определите на каких интервалах функции возрастают и на каких интервалах убывают:

1. $f(x) = x^x$.
2. $f(x) = x/(x + 1)$.
3. $f(x) = (x + 1)/x$.

4. Найдите локальные минимумы для следующих функций:

1. $f(x, y, z, u) = 2(x^2+y^2+z^2+u^2)-x(y+z-u) = yz-3x-8y-5z-9u$, начальная точка $\{1, 1, 1, 1\}$
2. $f(x, y, z) = 2x^2 + 2y^2 + z^2 - 2xy + yz - 7y - 4z$, начальные точки $\{1, 1, 1\}$, $\{0, 1, 0\}$, $\{1, 0, 1\}$, $\{0, 0, 1\}$
3. $f(x, y, z, u) = xyzu + 1/x + 1/y + 1/z + 1/u$, начальная точка $\{0.7, 0.7, 0.7, 0.7\}$

Глава 6

Приближение функций

6.1. Введение

В данной главе рассматриваются методы интерполяции функций и изучаются интерполяционные многочлены в форме Лагранжа и Ньютона, а также рассматривается сплайн–аппроксимация. Обсуждаются способы повышения точности аппроксимации. Приводятся формулы численного дифференциирования.

Пусть на отрезке $[a, b]$ задан набор точек, называемых узлами, x_i и соответствующие значения какой либо величины $f_i = f(x_i),\ i = 0, 1, \ldots, n$. Задача интерполяции состоит в построении приближающей функции $\varphi(x)$, такой, что $\varphi(x_i) = f_i, \forall i$. Функция $\varphi(x)$ должна давать достаточно "хорошие" значения между x_i и x_{i+1} и достаточно "хорошо" совпадать с приближаемой функцией $f(x)$. Критерии "хорошего" поведения функции различны в разных задачах и не существует его точного определения.

Определение 6.1 *Задача построения такой приближающей функции, при котором в узлах значения приближающей и приближаемой функций совпадают, называется лагранжевой интерполяцией.*

Такие задачи возникают в различных областях знаний. Например, пусть заданы n точек x_i, в которых известны экспериментальные значения функций $f(x_i)$. Если значения f_i представляют собой достаточно гладкую функцию, то эти значения можно интерполировать тоже достаточно гладкой функцией.

Основной целью интерполяции является получение хорошего и быстрого алгоритма для вычисления значений $\varphi(x)$ для тех x, которые ле-

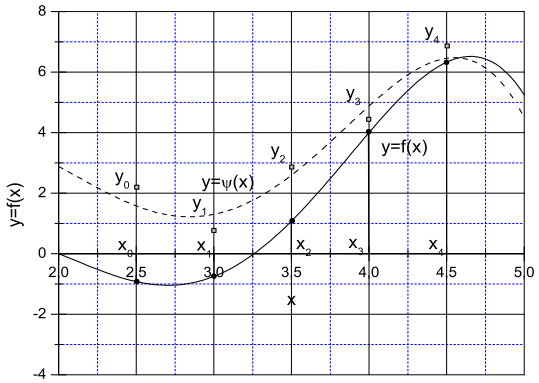

Рис. 6.1:

жат в промежутках между x_i и x_{i+1}. Для построения алгоритма необходимо знать, как должна вести себя функция между точками x_i и x_{i+1}.

На рисунке 6.1 приведены два способа аппроксимации функций. В первом из них функция $\varphi(x_i) = f(x_i)$ проходит через точки, отмеченные на рисунке закрашенными кружками в точках $x_0, x_1, \ldots, x_n$. Второй способ позволяет провести через точки, отмеченные квадратами, аппроксимирующую кривую таким образом, чтобы минимизировать сумму квадратов отклонений. Существует много критериев задания поведения функции между этими точками. Точки x_i могут быть интерполированы бесконечным множеством различных функций, поэтому необходимо иметь некоторый критерий выбора. В качестве таких критериев чаще всего выбирают простоту вычислений и гладкость аппроксимирующих функций $\varphi(x)$. Часто используются линейные комбинации элементарных функций. При этом, при использовании линейной комбинации одночленов вида $a_i x^i$, мы получаем полиномиальную аппроксимацию. Линейная аппроксимация тригометрических функций вида $\sin kx, \cos kx$, приводит к тригонометрическим полиномам и т.д.

Основная идея метода интерполяции состоит в следующем:

1. Необходимо определить класс интерполирующей функции

2. Задать критерий близости интерполируемых точек и приближающей функции. Если выбрано точное совпадение в узловых точках то такая интерполяция называется лагранжевой, если выбран критерий минимума суммы квадратов отклонений, то такая интерпо-

ляция называется методом наименьших квадратов и т.д.

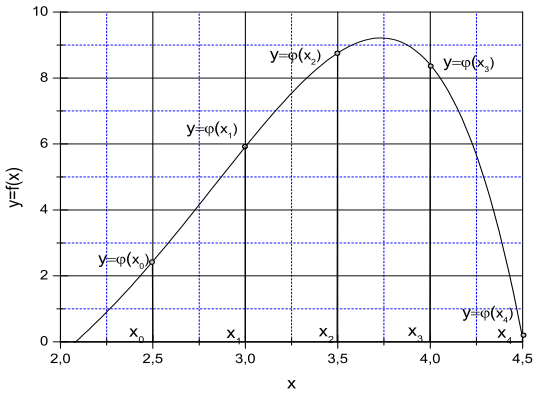

Рис. 6.2:

Зададим на отрезке [a,b] (рис.6.2) набор узлов $x_i, \quad i = 0, 1, \ldots, n$, в которых известны значения функций $\varphi(x_i)$. Эти значения могут быть получены из экспериментов или вычислены. В общем случае узлы могут быть неравноотстоящими. Примем, что аппроксимирующая функция $\varphi(x, c_0, c_1, \ldots, c_n)$ совпадает со значениями $f(x_i)$ во всех узлах x_i.

$$\varphi(x, c_0, c_1, \ldots, c_n) = f_i, \quad 0 \leq i \leq n \tag{6.1}$$

Такой способ введения аппроксимирующей функции называется **лагранжевой интерполяцией** (смотри **Определение 6.1**). Параметры c_i определяются из условия (6.1) и называются условиями Лагранжа. При линейной интерполяции аппроксимирующая функция ищется в виде линейной комбинации базисных функций $\varphi_i(x)$:

$$\varphi(x, c_0, c_1, \ldots, c_n) = \sum_{i=0}^{n} c_i \varphi_i(x) \tag{6.2}$$

Система (6.2) является линейно независимой, имеет единственное решение, если ее определитель $\neq 0$.

$$det = \begin{vmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \cdots & \varphi_n(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_n(x_1) \\ \cdots & \cdots & \cdots & \cdots \\ \varphi_0(x_n) & \varphi_1(x_n) & \cdots & \varphi_n(x_n) \end{vmatrix} \neq 0.$$

Параметры c_i можно найти, если в узлах аппроксимации подставить базисную функцию (6.2) в уравнение (6.1):

$$f_k = \sum_{i=0}^{n} c_i \varphi_i(x_k), \;\; k = 0, \ldots, n.$$

Как уже отмечалось выше, в зависимости от выбора базисных функций можно получить соответствующую интерполяцию. Если выбрать линейно независимую систему степенных функций $1, x, x^2, \ldots, x^n$, мы получим полиномиальную интерполяцию:

$$\varphi(x) = P_n(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n = \sum_{i=0}^{n} c_i x^i. \qquad (6.3)$$

Здесь c_i свободные параметры интерполяции (коэффициенты интерполяции). Если подставить значения x_i в (6.3) и использовать уравнение (6.1) получим систему алгебраических уравнений относительно c_i.

$$P_k(x) = f_k \quad k = 0, \ldots, n. \qquad (6.4)$$

Определитель системы отличен от нуля в случае несовпадающих узлов и называется определителем Вандермонда:

$$\begin{vmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{vmatrix} = \prod_{j>i\geq 0}^{n} (x_j - x_i) \neq 0.$$

Для того, чтобы показать то, что $P_n(x)$ является единственным полиномом, используем основную теорему алгебры. Эта теорема утверждает, что полином степени n имеет ровно n корней.

Теорема 6.1 *Алгебраическое уравнение $P_n(x) = 0$ n степени имеет ровно н корней, действительных или комплексных, при условии, что каждый корень считается столько раз, какова его кратность.*

Допустим, что это не так. Предположим, что существует другой полином степени n: $Q_n(x)$, который тоже проходит через эти же точки x_i. Обозначим через $R_n(x) = P_n(x) - Q_n(x)$ – разность между этими полиномами. Степень полинома $R_k(x) \leq n$ и $R_n(x_j) = P_n(x_j) - Q_n(x_j) = y_j - y_j = 0, \;\; (j = 0, 1, \ldots, n)$.

Отсюда следует, что $R_n(x) \equiv 0$ и $P_n(x) = 0, Q_n(x) = 0$. Таким образом решение системы линейных уравнений существует и единственно.

Это означает существование единственного интерполяционного полинома. Существует четыре способа интерполяции функции на отрезке $[a, b]$

1. Глобальный;
2. Локальный;
3. Кусочный;
4. Кусочно – глобальный (сплайн аппроксимация).

6.2. Интерполяционный полином Лагранжа

Как следует из вышесказанного, интерполирование предполагает вычисление неизвестных значений функций путем получения взвешенного среднего значения функций в известных соседних точках. При линейном интерполировании используется отрезок прямой, которая проходит через две соседние точки. Тангенс угла наклона отрезка между точками (x_0, y_0) и (x_1, y_1) к оси Ox равен

$$m = \frac{y_1 - y_0}{x_1 - x_0} \Rightarrow L(x) = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0}. \tag{6.5}$$

Из (6.5) следует, что мы получили полином степени $\leqslant 1$. Можно записать (6.5) в другом виде:

$$L_1(x) = y_0\frac{x - x_1}{x_0 - x_1} + y_1\frac{x - x_0}{x_1 - x_0}. \tag{6.6}$$

Члены вида $\frac{x-x_1}{x_0-x_1}$ называют лагранжевыми многочленами. Интерполяционный полином Лагранжа ищется в виде:

$$L_n(x) = \sum_{i=0}^{n} f_i l_i(x). \tag{6.7}$$

Из (6.6) следует, что при $x = x_0$ получаем 1, при $x = x_1$ получаем 0. Это означает, что лагранжевы многочлены $l_i(x)$ степени n удовлетворяют условию:

$$l_i(x_j) = \begin{cases} 1, & \text{если } i = j \\ 0, & \text{если } i \neq j \end{cases} \tag{6.8}$$

Таким образом, многочлены $l_i(x)$ имеют вид:

$$l_i(x) = c_i(x - x_0)(x - x_1)(x - x_2)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_n).$$

Отсюда, если подставить значения в узлах, получим:

$$c_i = \frac{1}{(x_i - x_0)(x_i - x_1)\dots(x_i - x_n)},$$

так как из (6.8) следует, что $l_i(x_i) = 1$. В общем виде лагранжевы полиномы можно записать в виде:

$$l_i(x) = \prod_{i=0, i\neq j}^{n} \frac{(x - x_j)}{(x_i - x_j)} \tag{6.9}$$

Если (6.9) подставить в (6.7), то интерполяционный полином Лагранжа можно записать в виде:

$$L_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, i\neq j}^{n} \frac{(x - x_j)}{(x_i - x_j)} \tag{6.10}$$

Интерполяционный полином Лагранжа позволяет найти значения функции в промежутках между известными значениями $f(x_i)$. Использование интерполяционной функции в виде полинома Лагранжа имеет как преимущества, так и недостатки. К достоинствам полинома Лагранжа можно отнести

1. Формула (6.10) справедлива как для равноотстоящих, так и для неравноотстоящих и несовпадающих узлов.

2. Интерполяционный полином Лагранжа удобен тем, что когда значения функций меняются, а узлы интерполяции неизменны – не требуется проводить лишних вычислений.

3. Общее число арифметических операций для вычисления интерполяционного полинома Лагранжа наименьшее и пропорционально n^2.

Недостатком интерполяции функция с помощью полиномов Лагранжа является то, что когда добавляются узлы интерполяции, приходится проводить все вычисления заново.

На практике часто встречается линейная и квадратичная интерполяция. Интерполяционный полином Лагранжа в случае линейной

$$L_1(x) = f_0 \frac{(x - x_1)}{(x_0 - x_1)} + f_1 \frac{(x - x_0)}{(x_1 - x_0)}, \tag{6.11}$$

и квадратичной интерполяции

$$L_2(x) = f_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}, \tag{6.12}$$

являются полиномами первой и второй степени соответственно.

Пример 6.1 Построить полиномы Лагранжа первой и второй степени, которые приближают функцию $y = x^3$ на интервале [-2,3]. На рисунке 6.3 функция $y = x^3$ изображена сплошной линий. Построим полином Лагранжа первой степени по точкам $x_0 = -1,\ x_1 = 1$. Значения функций в этих точках равны $f(x_0) = -1$ и $f(x_1) = 1$. Из уравнения (6.11) получим

$$L_1(x) = -1 \cdot \frac{x - 1}{-1 - 1} + 1 \cdot \frac{x + 1}{1 + 1} = 2x = 0.$$

Таким образом, полином Лагранжа первой степени $y = x$. На рисунке 6.3 изображен пунктирной линий. Полином Лагранжа второй степени получим из (6.12) по точкам $x_0 = 0,\ x_1 = 1,\ x_2 = 2$. Значения функций в этих точках равны $f(x_0) = 0,\ f(x_1) = 1,\ f(x_2) = 8$.

$$L_2(x) = 0 \cdot \frac{(x - 1)(x - 2)}{(0 - 1)(0 - 2)} + 1 \cdot \frac{(x - 0)(x - 2)}{(1 - 0)(1 - 2)} + 8 \cdot \frac{(x - 0)(x - 1)}{(2 - 0)(2 - 1)} = 3x^2 - 2x = 0.$$

Кривая, построенная по интерполяционному полиному Лагранжа первой степени проходит через две точки $x_0 = -1,\ x_1 = 1$. Полином Лагранжа второй степени проходит через три точки – $x_0 = 0,\ x_1 = 1,\ x_2 = 2$.

Пример 6.2. Рассмотрим $y = f(x) = \cos(x)$ на интервале [0,1.2]. Построим квадратичный интерполирующий полином $P_2(x)$ по трем узлам: $x_0 = 0.0;\ x_1 = 0.6;\ x_2 = 1.2$.

Подставим значения $x_0 = 0.0;\ x_1 = 0.6;\ x_2 = 1.2$ и соответствующие значения $f_0 = \cos(0.0) = 1,\ f_1 = \cos(0.6) = 0.825336,\ f_2 = \cos(1.2) = 0.362358$ в уравнение (6.12) и получим:

$$L_2(x) = f_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)},$$

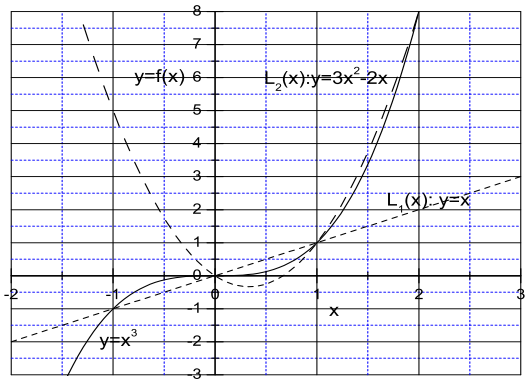

Рис. 6.3:

$$L_2(x) = 1.0\frac{(x-0.6)(x-1.2)}{(0.0-0.6)(0.0-1.2)} + 0.825336\frac{(x-0.0)(x-1.2)}{(0.6-0.0)(0.6-1.2)}$$
$$+0.362358 \cdot \frac{(x-0.0)\cdot(x-0.6)}{(1.2-0.0)\cdot(1.2-0.6)} = 1.388889 \cdot (x-0.6)\cdot(x-1.2)$$
$$-2.292599 \cdot (x-0.0)\cdot(x-1.2) + 0.503275 \cdot (x-0.0)\cdot(x-0.6) =$$
$$-0.3981x^2 - 0.05382x + 1.$$

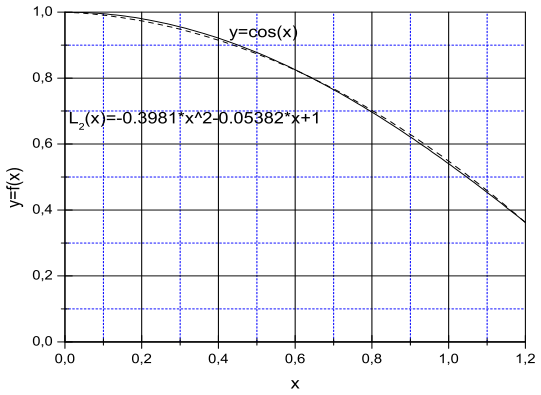

Рис. 6.4:

На рисунке 6.4 сплошной линией изображен график $y = \cos(x)$, пунктирной линий интерполяционный полином второй степени $y = -0.3981x^2 - 0.05382x + 1$, который на отрезке [0,1.2] интерполирует функцию.

Задание: Аналогично квадратичному попробуйте построить кубический интерполирующий полином $P_3(x)$ по четырем узлам: $x_0 = 0.0$; $x_1 = 0.4$; $x_2 = 0.6$ $x_3 = 1.2$

Упражнения

I. Постройте полиномы Лагранжа, которые приближают функцию $f(x) = x^3$ используя:

1. линейный интерполяционный полином, для узлов $x_0 = -1$ и $x_1 = 0$.
2. квадратичный интерполяционный полином, для узлов $x_0 = -1$, $x_1 = 0$ и $x_2 = 1$.
3. кубический интерполяционный полином, для узлов $x_0 = -1$, $x_1 = 0$, $x_2 = 1$ и $x_3 = 2$.
4. квадратичный интерполяционный полином, для узлов $x_0 = 0$, $x_1 = 1$ и $x_2 = 3$.

II. Пусть $f(x) = 2\sin(\pi/6 \cdot x)$, где x задана в радианах.

1. Используйте квадратичную интерполяцию Лагранжа, построенную по узлам $x_0 = 0$, $x_1 = 1$ и $x_2 = 3$, чтобы найти приближения к функции в узлах $f(4.0)$ и $f(3.5)$.

6.2.1. Ошибки интерполяции

Рассмотрим ошибки, которые возникают при интерполяции непрерывной функции $f(x)$ полиномами Лагранжа. Обозначим через $R_n(x)$ ошибку интерполяции. Тогда эта погрешность может быть представлена в виде:

$$R_n(x) = f(x) - L_n(x). \tag{6.13}$$

Пусть $f(x) \in C_{n+1}[a, b]$.

Как показано в [14], при интерполяции функции, заданной на неравномерной сетке интерполяционным полиномом Лагранжа:

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

для любого $\xi \in [x_0, x_n]$ возникает погрешность:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\omega_n(x),$$

$\omega_n(x) = (x - x_0)(x - x_1)\dots(x - x_n)$ Точное значение $R_n(x)$ найти нельзя из-за неопределенности ξ [15]. Поэтому можно оценить погрешность в некоторой произвольной точке $\xi \in [x_0, x_n]$:

$$|f(\xi) - L_n(\xi)| \le \frac{M_{n+1}}{(n+1)!}|\omega_n(\xi)|,$$

здесь $M_{n+1} = \max\limits_{[a,b]}|f^{(n+1)}(x)|$.

Максимальную погрешность в любой точке можно оценить следующим образом:

$$|f(x) - L_n(x)| \le \frac{M_{n+1}}{(n+1)!}|\omega_n(x)|.$$

6.3. Интерполяционный полином Ньютона

Существуют и другие способы нахождения интерполяционных полиномов. Рассмотрим интерполяционный полином, записанный в форме Ньютона. Запись полинома в этой форме имеет некоторые преимущества, по сравнению с записью в форме Лагранжа. При использовании полиномов Лагранжа каждый полином строится индивидуально и мы не можем получить соотношения между $P_{n-1}(x)$ и $P_n(x)$. Построим полином Ньютона, который будет обладать реккурентными свойствами. Будем строить полином таким образом, чтобы в узлах интерполяции он совпадал со значениями функции $f(x)$.

$$P_1(x) = a_0 + a_1(x - x_0), \tag{6.14}$$
$$P_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1). \tag{6.15}$$

В общем случае интерполяционный полином Ньютона получается при помощи рекуррентного соотношения:

$$P_n(x) = P_{n-1}(x) + a_n(x - x_0)(x - x_1)(x - x_2)\dots(x - x_{n-1}). \tag{6.16}$$

Полином (6.16) называется интерполяционным полиномом Ньютона, заданном в n+1 узле.

Пример 6.3 Заданы узлы интерполяции $x_0 = 1,\ x_1 = 3\ x_2 = 4$ и коэффициенты $a_0 = 5,\ a_1 = -2,\ a_2 = 0.5,\ a_3 = -0.1,\ a_4 = 0.003$. Необходимо определить полиномы $P_i(x)$ и вычислить $P_i(2.5)$ для i=1,2,3,4. Из (6.14) получим

$$\begin{aligned} P_1(x) &= 5 - 2(x-1), \\ P_2(x) &= 5 - 2(x-1) + 0.5(x-1)(x-3), \\ P_3(x) &= P_2(x) - 0.1(x-1)(x-3)(x-4), \\ P_4(x) &= P_3(x) + 0.003(x-1)(x-3)(x-4)(x-4). \end{aligned} \tag{6.17}$$

Вычислим значения полиномов в точке x=2.5: $P_1(2.5) = 2,\ P_2(2.5) = 1.625,\ P_3(2.5) = 1.5125,\ P_4(2.5) = 1.50575$.

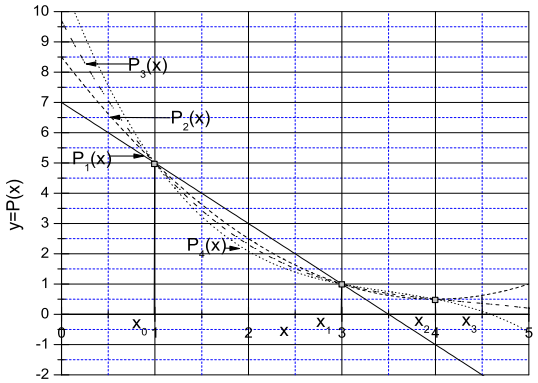

Рис. 6.5:

На рисунке 6.5 приведены четыре полинома Ньютона $P_1(x),\ P_2(x),\ P_3(x),\ P_4(x)$ (6.17). Отметим, что полиномы всех степеней проходят через точки x_0, x_1, полиномы второй, третьей и четвертой степеней проходят через точки x_0, x_1, x_2.

Введем понятие разделенных разностей. Будем называть разделенной разностью следующее соотношение:

$$f(x_i, x_j) = \frac{f(x_i) - f(x_j)}{x_i - x_j}.$$

Разделенные разности второго порядка определяются так:

$$f(x_i, x_j, x_k) = \frac{f(x_i, x_j) - f(x_j, x_k)}{x_i - x_k}.$$

Рекуррентное соотношение для разделенных разностей n-го порядка можно записать в виде:

$$f(x_i, x_j, x_k, \ldots, x_{n-1}, x_n) = \frac{f(x_i, x_j, x_k, \ldots, x_{n-1}) - f(x_j, x_k, \ldots, x_n)}{(x_i - x_n)}.$$

Теорема 6.1 (полином Ньютона). *Предположим, что* $x_0, x_1, x_2, x_3 \ldots, x_n$ *представляют собой* $n+1$ *различных чисел, принимающих значения на интервале [a,b]. Тогда существует единственный полином* $P_n(x)$ *степени не более чем* n*, обладающий следующим свойством:*

$$f(x_j) = P_n(x_j), \quad j = 0, 1, \ldots, n.$$

Полином Ньютона можно записать в виде:

$$P_n(x) = a_0 + a_1(x - x_0) + \ldots + a_n(x - x_0)(x - x_1)\ldots(x - x_{n-1}),$$

где $a_k = f(x_0, x_1, \ldots, x_{n-1})$.

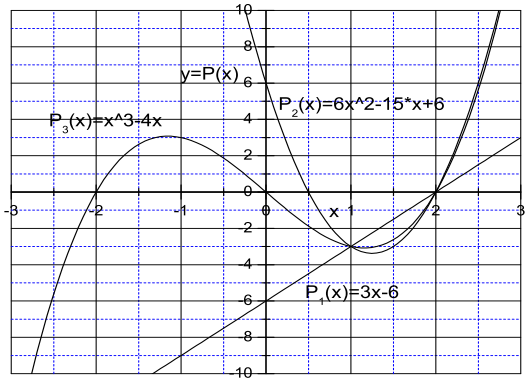

Рис. 6.6:

Пример 6.4 Построим разделенные разности для уравнения $f(x) = x^3 - 4x$ в узлах $x_0 = 1,\ x_1 = 2,\ x_2 = 3, \ldots, x_5 = 6$. Найдем полиномы Ньютона $P_1(x), P_2(x), P_3(x)$, основанные на точках x_0, x_1, x_2, x_3. Из (6.16) найдем коэффициенты полинома Ньютона $P_3(x)$ $a_0 = -3,\ a_1 = 3,\ a_2 = 6,\ a_3 = 1$. Окончательно, первые три полинома Ньютона можно записать следующим образом:

$$P_1(x) = -3 + 3(x-1) = 3x - 6,$$

$$P_2(x) = -3 + 3(x-1) + 6(x-1)(x-2) = 6x^2 - 15x + 6,$$

$$P_3(x) = -3 + 3(x-1) + 6(x-1)(x-2) + (x-1)(x-2)(x-3) = x^3 - 4x.$$

Полином Ньютона третьей степени совпадает с аппроксимируемой функцией. Все три полинома имеют две общие точки $x_0 = 1,\ x_1 = 2$. На рисунке 6.6 приведены графики функций $P_1(x), P_2(x), P_3(x)$. Обозначения показаны на рисунке.

Программа 6.1 позволяет вычислить коэффициенты интерполирующего полинома. В первом цикле программы задается шаг x и вычисляются значения заданной функции в этой точке. Функция задается в главной программе, после оператора

```
contains.
```

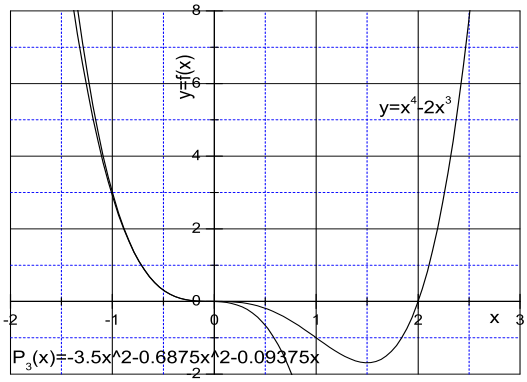

Рис. 6.7:

Пример 6.5. Вычислим интерполирующие полиномы третьего и четвертого порядка для функции:

$$y = x^4 - 2x^3, \tag{6.18}$$

на отрезке [-1,1]. Если в программе 6.1 задать nx=5 это дает пять коэффициентов полинома 4 порядка. Мы получим коэффициенты почти совпадающие с коэффициентами исходного полинома (6.18).

```
coeff is equal to 0.999998   -2.000003   -0.000001 0.000000
0.000000
```

Если задать nx=4 мы получим коэффициенты кубического полинома:

```
 coeff is equal to
    -3.500000   -0.687500   -0.093750    0.000000
```

$$y = -3.5x^3 - 0.6875x^2 - 0.09375x. \tag{6.19}$$

На рисунке 6.7 приведены графики уравнений (6.18) и (6.19). Отметим, что на отрезке [-1.5,0] полином третьей степени достаточно хорошо аппроксимирует исходную функцию (6.18). Программа 6.1 вычисляет ошибки, которые получаются при этом.

Программа 6.1 Программа вычисления коэффициентов интерполирующего полинома

```
 program Interpol
!  Программа вычисления коэффициентов интерполирующего полинома
      integer,parameter:: nx=5
      real,parameter:: dx=1.0
      integer:: i,j
      real g,sum,x,x1(nx),y1(nx),coeff(nx)
         write(*,*) 'Function y=x**4-2x**3 from -1 to 1'
          do i=1,nx
            x1(i)=-1 + i*dx/nx
            y1(i)=f(x1(i))
          enddo
        call lcoeff(x1,y1,nx,coeff)
        write(*,*) '    coeff is equal to'
        write(*,'(1x,6f12.6)') (coeff(i),i=1,nx)
        write(*,'(1x,t10,a1,t20,a4,t29,a10)') &
```

```
        'x','f(x)','polynomial'
   do i=1,11
     x=-1.4+i/2.50
     g=f(x)
     sum=coeff(nx)
     do j=nx-1,1,-1
       sum=coeff(j)+sum*x
     enddo
     write(*,'(1x,3f12.6)') x,g,sum
   enddo
   write(*,*) '-------------------------------------'
  contains
  function f(x)
    real f,x
    f = x**4-2.0*x**3
  end function f
 end program Interpol
     subroutine lcoeff(xa,ya,n,cof)
 integer,parameter:: max = 10
 integer:: i,j,k,n
 real cof(n),xa(n),ya(n)
 real dy,xmin,x(max),y(max)
do i=1,n
   x(i)=xa(i)
   y(i)=ya(i)
enddo
 do j=1,n
   call interpolation(x,y,n+1-j,0.,cof(j),dy)
   xmin=1.e38
   k=0
   do i=1,n+1-j
     if (abs(x(i)).lt.xmin)then
```

```
        xmin=abs(x(i))
        k=i
      endif
      if(x(i).ne.0.)y(i)=(y(i)-cof(j))/x(i)
    enddo
    do i=k+1,n+1-j
      y(i-1)=y(i)
      x(i-1)=x(i)
    enddo
  enddo
  end subroutine lcoeff
     subroutine interpolation(xa,ya,n,x,y,dy)
  integer,parameter::nx=10
  integer n,i,m,ns
  real:: dy,x,y,xa(n),ya(n)
  real contrl,dif,dift,ho,hp,w,c(nx),d(nx)
  ns=1
  dif=abs(x-xa(1))
do i=1,n
   dift=abs(x-xa(i))
  if (dift.lt.dif) then
     ns=i
     dif=dift
  endif
    c(i)=ya(i)
    d(i)=ya(i)
enddo
  y=ya(ns)
  ns=ns-1
do m=1,n-1
    do i=1,n-m
      ho=xa(i)-x
```

```
      hp=xa(i+m)-x
      w=c(i+1)-d(i)
      contrl=ho-hp
     if(contrl.eq.0.) then
      write(*,*) 'ERROR in interpolation'
      stop
     endif
      contrl=w/contrl
      d(i)=hp*contrl
      c(i)=ho*contrl
  enddo
    if (2*ns.lt.n-m)then
      dy=c(ns+1)
    else
      dy=d(ns)
      ns=ns-1
    endif
    y=y+dy
  enddo
  end subroutine interpolation
```

На рисунке 6.8 показана исходная функция $y = \exp(x)$ и аппроксимирующий полином

$$y = 0.076437x^4 + 0.123288x^3 + 0.524018x^2 + 0.994x + 1. \qquad (6.20)$$

Упражнения

1. Постройте таблицу разностных отношений для функций, определенных в таблицах.

2. Постройте полиномы Ньютона $P_1(x),\ P_2(x),\ P_3(x)$

3. Вычислите полиномы Ньютона для 2 пункта, в заданных точках x.

$$f(x) = x^{\frac{1}{2}},\ x = 4.5,\ 7.5.$$

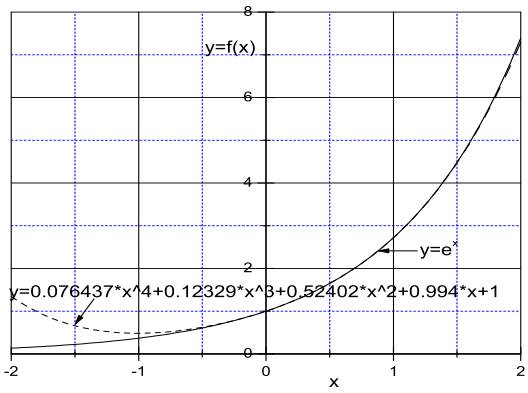

Рис. 6.8:

k	x_k	$f(x_k)$
0	4.0	2.00000
1	5.0	2.23607
2	6.0	2.44949
3	7.0	2.64575
4	8.0	2.82843

$$f(x) = 3.6/x,\ x = 2.5,\ 3.5.$$

k	x_k	$f(x_k)$
0	1.0	3.60
1	2.0	1.80
2	3.0	1.20
3	4.0	0.90
4	5.0	0.72

6.4. Интерполяционные полиномы Чебышёва

Прменение многочленов высокого порядка не дает требуемой точности и лучшего приближения к исходной функции. На рисунке 6.9 приведен известный пример аппроксимации функции Рунге. Эта функция задается

уравнением (6.21):

$$y = \frac{1}{(1 + 25x^2)}. \tag{6.21}$$

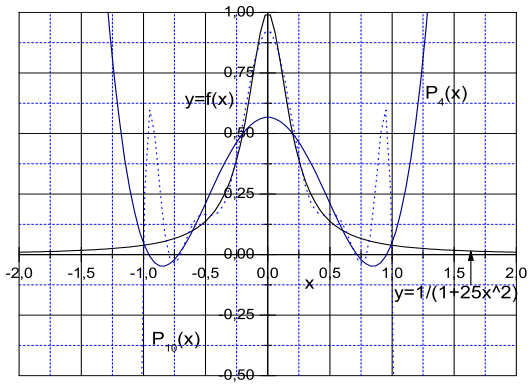

Рис. 6.9:

Построим полиномы четвертой степени (6.22) и десятой степени (6.23). На рисунке 6.9 обозначены как $P_4(x)$ и $P_{10}(x)$ соответственно.

$$P_4(x) = 1.201924x^4 - 1.72077x^2 + 0.567308 \tag{6.22}$$

$$\begin{aligned} P_{10}(x) = -89.497597x^{10} + 215.119507x^8 - 184.304993x^6 \\ +69.760551x^4 - 11.962015x^2 + 0.922965. \end{aligned} \tag{6.23}$$

При увеличении степени интерполирующего полинома, центральная часть функции Рунге (внутри отрезка [-0.725,0.725]) интерполируется лучше, но вне этого отрезка возникают осцилляции. Эти колебания увеличиваются с увеличением степени полинома.

П.Л. Чебышёв построил совокупность узлов интерполяции таким образом, чтобы минимизировать ошибки интерполяции. Узлы интерполяции располагаются неравномерно, в нулях полинома Чебышёва. Эти полиномы получаются рекуррентно по следующему алгоритму. Первый многочлен $T_0(x) = 1$, второй $T_1(x) = x$. Далее используется рекуррентное соотношение

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x). \tag{6.24}$$

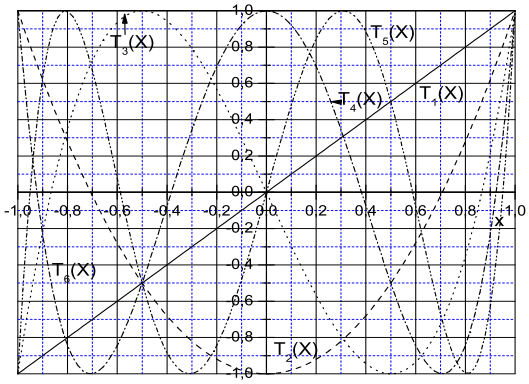

Рис. 6.10:

В таблице 6.1 приведены первые 8 полиномов Чебышёва.

Таблица 6.1

$T_i(x)$	полином
$T_0(x) =$	1
$T_1(x) =$	x
$T_2(x) =$	$2x^2 - 1$
$T_3(x) =$	$4x^3 - 3x$
$T_4(x) =$	$8x^4 - 8x^2 + 1$
$T_5(x) =$	$16x^5 - 20x^3 + 5x$
$T_6(x) =$	$32x^6 - 48x^4 + 18x^2 - 1$
$T_7(x) =$	$64x^7 - 112x^5 + 56x^3 - 7x$

На рисунке 6.10 изображены первые шесть полиномов Чебышёва. Полином $T_n(x)$ имеет n корней на отрезке [-1,1]. Эти корни находятся в точках:

$$x = \cos(\frac{\pi(k - \frac{1}{2})}{n}) \; k = 1, 2, \ldots, n \tag{6.25}$$

На этом же интервале находится n+1 минимумов и максимумов. все минимальные значения равны -1, максимальные равны +1. Полиномы Чебышёва ортогональны на отрезке [-1,1]. С помощью полиномов Чебышёва можно минимизировать ошибку $R_n(x)$ (6.13).

$$R_n(x) = \omega_n(x)\frac{f^{(n+1)}(\xi)}{(n+1)!}, \tag{6.26}$$

где

$$\omega_n(x) = (x - x_0)(x - x_1)\dots(x - x_n). \tag{6.27}$$

Из (6.26) можно сформировать максимальное значение $|\omega_n(x)|$, $\forall x \in [-1, 1]$, умноженного на $\max|\frac{f^{(n+1)}(\xi)}{(n+1)!}|$. Для минимизации этого значения Чебышёв предложил выбирать такие значения $x_0, x_1, \dots, x_n$, чтобы $\omega_n(x) = (1/2^n) \cdot T_{n+1}(x)$. В теореме 6.2 приводится утверждение того, что такой полином единственный.

Теорема 6.2 *Предположим, что n фиксировано. Среди всех возможных вариантов выбора для $\omega(x)$ в (6.27) и среди всех возможных вариантов выбора различных узлов $\{x_k\}_{k=0}^n$ на интервале [-1,1] полином $T(x) = T_{n+1}(x)/2^n$ является единственным, который обладает следующим свойством:*

$$\max_{-1 \leqslant x \leqslant 1} \{|T(x)|\} \leqslant \max_{-1 \leqslant x \leqslant 1} \{|\omega(x)|\}. \tag{6.28}$$

Следствие.

$$\max_{-1 \leqslant x \leqslant 1} \{|T(x)|\} = \frac{1}{2^n}. \tag{6.29}$$

Для произвольной функции $f(x)$ на интервале [-1,1] коэффициенты c_j интерполяционных полиномов Чебышёва определяются так:

$$c_j = \frac{2}{n} \sum_{k=1}^{n} f(x_k) T_{j-1}(x_k) = \tag{6.30}$$

$$\frac{2}{n} \sum_{k=1}^{n} f\Big[\cos\Big(\frac{\pi(k - 1/2)}{n}\Big)\Big] \cdot \cos\Big(\frac{\pi(j-1)(k-1/2)}{n}\Big).$$

Для аппроксимации $f(x)$ используют приближенную формулу:

$$f(x) \approx \Big[\sum_{k=1}^{n} c_k T_{k-1}(x)\Big] - \frac{1}{2} c_1, \tag{6.31}$$

где коэффициенты c_j вычисляются по (6.30). Для функции Рунге (6.21) аппроксимирующая функция с одиннадцатью коэффициентами Чебышёва имеет вид:

$$f(x) \approx -0.053782 \cdot T_{10}(x) + 0.080024 \cdot T_8(x) - 0.119070 \cdot T_6(x)$$
$$+0.177167 \cdot T_4(x) - 0.263611 \cdot T_2(x) + 0.392232.$$

На рисунке 6.11 изображен график функции Рунге (6.21) и график аппроксимирующей функции, построенной с помощью полиномов Чебышёва.

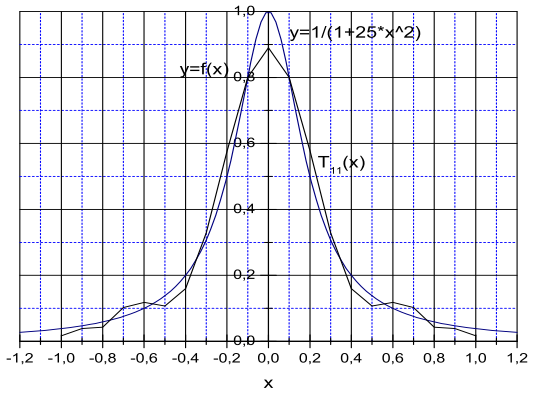

Рис. 6.11:

Отметим, что колебания, которые были при аппроксимации функции полиномами Лагранжа и Ньютона (рис. 6.9) практически отсутствуют.

6.5. Интерполяция сплайнами

Построение интерполяционных полиномов Лагранжа и Ньютона достаточно эффективно для не очень большого количества узлов. Если количество узлов велико, построенные по ним полиномы не дают удовлетворительных результатов. Как отмечалось выше, для n узлов, интерполяционный полином имеет n-1 локальный максимум и минимум. Это приводит к большим колебаниям полученной аппроксимирующей кривой и большому отклонению от значений интерполируемой функции. Кроме использования полиномов Чебышёва, для устранения этих колебаний используется сплайн-интерполяция. Суть данной интерполяции заключается в построении такого кусочного полинома, который проходит через узлы интерполяции, а между узлами необходимо "сшивать" значения функций.

Определение 6.2 *Кусочно-непрерывный многочлен n степени, который в узлах аппроксимации принимает значения интерполируемой*

функции и непрерывен вместе с производной до (n-1) порядка на отрезке интерполяции называется **сплайном**.

$$S(x) = \sum_{k=0}^{n} a_{i,k} x^k, \quad x_{i-1} \le x \le x_i; \tag{6.32}$$

Максимальная степень полинома (6.32) называется *степенью сплайна*. Разница между степенью сплайна и порядком наивысшей непрерывной производной называется *дефектом сплайна*.

Наиболее простым способом является построение кусочно-линейного интерполяционного полинома. Он состоит из отрезков прямых, которые проходят через узлы интерполяции. Используем полином Лагранжа для построения такой кусочно-линейной интерполяции.

$$S_k(x) = y_k \frac{x - x_{k+1}}{x_k - x_{k+1}} + y_{k+1} \frac{x - x_k}{x_{k+1} - x_k}, \text{ для } x_k \le x \le x_{k+1}.$$

Можно получить аналогичное выражение, если использовать тангенс угла наклона линии в данной точке:

$$S_k(x) = y_k + d_k(x - x_k),$$

где $d_k = (y_{k+1} - y_k)/(x_{k+1} - x_k)$. Таким образом линейный сплайн можно записать в виде:

$$S(x) = \begin{cases} y_0 + d_0(x - x_0), & x \subset [x_0, x_1] \\ y_1 + d_1(x - x_1), & x \subset [x_1, x_2] \\ \quad \vdots & \quad \vdots \\ y_k + d_k(x - x_k), & x \subset [x_k, x_{k+1}] \\ \quad \vdots & \quad \vdots \\ y_{n-1} + d_{n-1}(x - x_{n-1}), & x \subset [x_{n-1}, x_n] \end{cases} \tag{6.33}$$

Более распространеным на практике является кубический сплайн.

Определение 6.3 кубического сплайна *Пусть на отрезке [a,b] существует n+1 точка: $a = x_0 < x_1 < \ldots < x_n = b$. Функция $S(x)$ называется* ***кубическим сплайном****, если существует n кубических полиномов $S_k(x)$, с коэффициентами $a_{k,0}, a_{k,1}, a_{k,2}, a_{k,3}$, которые удовлетворяют следующим условиям:*

1. $S(x) = S_k(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2 + d_k(x - x_k)^3$,
   для $x \in [_k, x_{k+1}]$, $k = 0, 1, \ldots, n - 1$,

2. $S_k(x_k) = y_k$, k=0,1, ... , n,
3. $S_k(x_{k+1}) = S_{k+1}(x_{k+1})$, k=0,1, ... , n-2,
4. $S'_k(x_{k+1}) = S'_{k+1}(x_{k+1})$, k=0,1, ... , n-2,
5. $S''_k(x_{k+1}) = S''_{k+1}(x_{k+1})$, k=0,1, ... , n-2.

Условия 1-5 определяют следующее.

- Первое условие утверждает, что $S(x)$ состоит из кубических полиномов.
- Во втором условии утверждается, что интерполирование осуществляется на множестве узлов $x_0, x_1, \ldots, x_n$.
- Условие три и четыре задают гладкость и непрерывность сплайнов.
- Условие пять задает непрерывность второй производной.

Построить кубический сплайн для n узлов можно определив $4 \cdot n$ неизвестных коэффициентов, определенных в свойстве 1. Обозначим $h_k = x_k - x_{i-k}$. Тогда, так как в узлах сетки должно выполняться свойство 2, получим:

$$S(x_{k-1}) = a_k = y_{k-1}, \tag{6.34}$$
$$S(x_k) = a_k + b_k h_k + c_k h_k^2 + d_k h_k^3. \tag{6.35}$$

Система (6.34) содержит $2 \cdot n$ уравнений. Недостающие уравнения можно получить, использовав свойства 3, 4 и 5.

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

Отсюда получаются еще $2(n-1)$ уравнений:

$$b_{k+1} = b_k + 2c_k h_k + 3d_k h_k^2, \tag{6.36}$$
$$c_{k+1} = c_k + 3d_k h_k. \tag{6.37}$$

Таким образом мы имеем $4n - 2$ уравнения. Для нахождения недостающих уравнений используем граничные условия на концах отрезка интерполяции. Существует несколько способов задания этих граничных условий. Первый способ – из каких либо условий известны граничные условия. Тогда задаем $S'(x_0) = \varphi(x_0)$, $S'(x_n) = \psi(x_n)$, где φ и ψ известные функции.

Если на границах условия неизвестны, можно считать, что кривизна сплайна там нулевая. Это дает

$$S''(x_0) = c_k = 0, \tag{6.38}$$

$$S''(x_n) = c_n + 3d_n h_n = 0. \tag{6.39}$$

Можно, в граничных точках, воспользоваться экстраполированием значений $S''(x_0)$ и $S''(x_n)$

Использовав необходимые условия для определения граничных условий, мы получили замкнутую систему уравнений для определения неизвестных a_k, b_k, c_k, d_k. Коэффициент a_k можно получить непосредственно из (6.34). Из уравнений (6.36) – (6.38) следует:

$$d_k = (c_{k+1} - c_k)/3h_k, \quad k = 1, 2, \dots, n \tag{6.40}$$

$$d_n = -c_n/3h_n. \tag{6.41}$$

Подставляя (6.40) и (6.41) в (6.34) и заменяя a_k на f_{k-1} получим

$$b_k = [(f_k - f_{k-1})/h_k] - 1/3h_k(c_{k+1} + 2c_k) \tag{6.42}$$

$$d_n = [(f_n - f_{n-1})/h_n] - 2/3h_n c_n. \tag{6.43}$$

Из уравнений (6.36)

$$b_{k+1} = b_k + h_k(c_k + c_{k+1})$$

или

$$b_k = b_{k-1} + h_{k-1}(c_{k-1} + c_k) \tag{6.44}$$

Из (6.42) и (6.43) получаем $c_{n+1} = 0$. Поставляя (6.42) и (6.43) в (6.44) получаем систему уравнений относительно c_k:

$$\begin{aligned} &c_1 = 0, \\ &h_{k-1}c_{k-1} + 2(h_{k-1} + h_k)c_k + h_k c_{k+1} = \\ &= 3[(f_k - f_{k-1})/h_k - (f_{k-1} - f_{k-2})/h_{k-1}], \quad k = 2, \dots, n \\ &c_n = 0. \end{aligned} \tag{6.45}$$

Таким образом получилась трехдиагональная система уравнений. Такие системы лучше всего решать методом прогонки (§3.3).
На рисунке 6.12 изображена функция

$$y = 1.7x^4 + x^3 - 3x^2 + 0.5x,$$

и кубический сплайн, который построен по точкам. Функция изображена треугольниками, кубический сплайн – сплошной линией. Нужно отметить хорошее совпадение между функцией и сплайн – аппроксимацией.

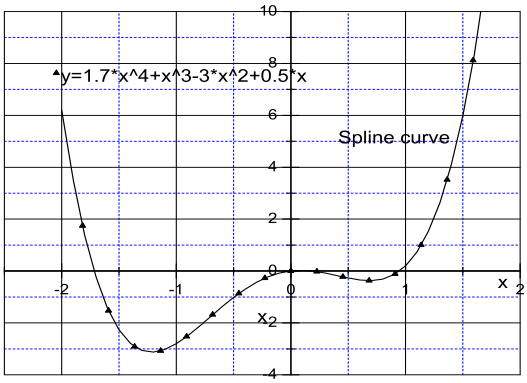

Рис. 6.12:

6.6. Метод наименьших квадратов

В различных областях науки и техники, в результате проведения экспериментов, получают совокупность экспериментальных точек $x_1, x_2, \ldots, x_n$ и соответствующих значений функций $y_1, y_2, \ldots, y_n$. Полученные значения найдены с некоторой погрешностью. Поэтому построение интерполяционного полинома, который проходит через эти точки, не дает хороших результатов. Так как экспериментальные данные определяются с ошибками, можно записать

$$f(x_k) = y_k + \varepsilon_k \tag{6.46}$$

где ε - ошибка измерения. Определим несколько норм, который позволяет определить, насколько далеко кривая (6.46) лежит от экспериментальных даных.

Максимальная ошибка: $R_\infty(f) = \max\limits_{1\le k\le n}\{|f(x_k) - y_k|\}$,

Средняя ошибка: $R_1(f) = \frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|$,

Среднеквадратичная ошибка: $R_2(f) = (\frac{1}{n}\sum_{k=1}^{n} |f(x_k) - y_k|^2)^{1/2}$.

Пример 6.6 Сравним максимальную, среднюю и среднеквадратичную ошибки для линейной функции $y = 8.6 - 1.6x$. Значения экспериментальных точек следующие:

$$(-1, 10), (0, 9), (1, 7), (2, 5), (3, 4), (4, 3), (5, 0), (6, -1).$$

Найдем значения функции $f(x_k)$ и e_k:

$$R_\infty(f) = \max\{0,2; 0,4; 0,0; 0,4; 0,2; 0,8; 0,6; 0,0\} = 0.8, \quad (6.47)$$

$$R_1(f) = \frac{1}{8}(2.6) = 0.325, \quad (6.48)$$

$$R_2(f) = (\frac{1.4}{8})^{1/2} \approx 0.41833 \quad (6.49)$$

Видно, что максимальная ошибка – наибольшая. Наилучшее приближение дает функция, которая определяется путем минимизации ошибок.

Задача построения приближающей функции методом наименьших квадратов формулируется следующим образом. Пусть заданы n точек x_k и значения функций в этих точках $y_k(x_k) = f_k$. Необходимо найти полином степени $F_n(x) = \sum_{i=0}^{n} a_i x^i$, который имеет минимальное квадратичное отклонение:

$$\Phi = \sum_{j=0}^{N} \left[F_n(x_j) - f_j\right]^2. \quad (6.50)$$

В зависимости от выбора показателя степени многочлена N, существуют линейные, квадратичные, кубические и полиномы более высоких степеней, которые позволяют провести аппроксимирующую кривую, удовлетворяющие условию (6.50). Рассмотрим построение квадратичного полинома методом наименьших квадратов. Для этого рассмотрим следующую теорему.

Теорема 6.3 *Предположим, что* $\{(x_k, y_k)\}_{k=1}^{N}$ *–N точек, абсциссы которых различны. Коэффициенты параболы, построенной методом наименьших квадратов*

$$y = f(x) = Ax^2 + Bx + C, \quad (6.51)$$

являются решениями линейной системы уравнений

$$\begin{aligned} A\Big(\sum_{j=0}^{N} x_k^4\Big) + B\Big(\sum_{j=0}^{N} x_k^3\Big) + C\Big(\sum_{j=0}^{N} x_k^2\Big) &= \sum_{j=0}^{N} y_k x_k^2, \\ A\Big(\sum_{j=0}^{N} x_k^3\Big) + B\Big(\sum_{j=0}^{N} x_k^2\Big) + C\Big(\sum_{j=0}^{N} x_k\Big) &= \sum_{j=0}^{N} y_k x_k, \\ A\Big(\sum_{j=0}^{N} x_k^2\Big) + B\Big(\sum_{j=0}^{N} x_k\Big) + CN &= \sum_{j=0}^{N} y_k. \end{aligned} \quad (6.52)$$

Найденные значения A,B и C позволяют построить параболу, которая обладает наименьшим значением квадратов отклонений построенной функции от заданных точек. Методом наименьших квадратов аналогично (6.52) можно построить аппроксимирующие полиномы более высокого порядка.

Пример 6.5. Построить полином второй степени методом наименьших квадратов по точкам (-2,-1), (1,1), (4,3) и (7,4). Подставим эти значения в систему уравнений (6.52).

$$\begin{aligned} 2674A + 400B + 70C &= 241, \\ 400A + 70B + 10C &= 43, \\ 70A + 10B + 4C &= 7. \end{aligned} \tag{6.53}$$

Решением системы линейных уравнений (6.52) является вектор $\{-0.0278,\ 0.7056,\ 0.47222\}$. Что дает квадратное уравнение:

$$-0.0278x^2 + 0.7056x + 0.47222 = 0. \tag{6.54}$$

На рисунке 6.13 треугольниками изображены экспериментальные точки, а сплошной линией квадратный полином (6.54), найденный методом наименьших квадратов.

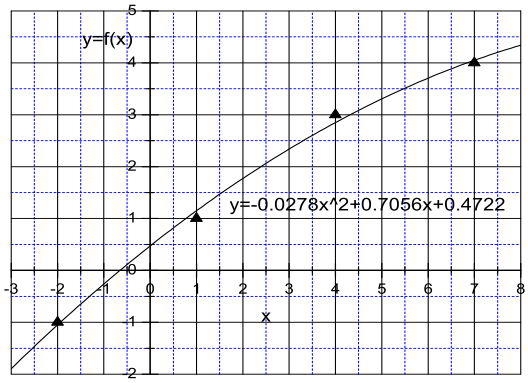

Рис. 6.13:

6.7. Упражнения

1. Найти полином Чебышёва $T_3(x)$, который приближает функцию $f(x) = \exp(x)$, на отрезке [-1,1].

2. Найти полином Чебышёва $T_n(x)$, который приближает функцию на отрезке $f(x) = \sqrt{x+2}$ [-1,1] для n=4.

3. Найти полином Чебышёва $T_n(x)$, который приближает функцию на отрезке $f(x) = \sqrt{x+2}$ [-1,1] для n=5.

4. Найти полином Чебышёва $T_n(x)$, который приближает функцию на отрезке $f(x) = \sqrt{x+2}$ [-1,1] для n=6.

5. Найти полином Чебышёва $T_n(x)$, который приближает функцию на отрезке $f(x) = x + 2^{(x+2)}$ [-1,1] для n=5.

6. Задана последовательность точек:

$$x_i = \{-1.5, -0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5\}$$

и значений функций в этих точках:

$$y_i = \{11.0, 10, 0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0\}.$$

Постройте по этим точкам

- полином первой степени,
- полином второй степени,
- полином третьей степени.

Глава 7

Численное дифференциирование и интегрирование

7.1. Введение

При решении задач, связанных с решением обыкновенных дифференциальных уравнений и дифференциальных уравнений в частных производных возникают проблемы, связанные с численным дифференциированием функций. Такие же проблемы возникают в тех случаях, когда необходимо продифференциировать функцию, заданную таблично, или достаточно сложную аналитическую функцию. Во всех таких случаях можно построить полином, который аппроксимирует функцию, и затем найти производные для этого полинома. Другими словами мы заменяем дифференциирование сложной функции, дифференциированием аппроксимирующей ее более простой полиномиальной функцией. Рассмотрим, для примера, дифференциирование функции Бесселя $J_1(x)$. Функцию Бесселя целого аргумента можно представить в виде:

$$J_n(x) = (\frac{1}{2}x^n) \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(n+k+1)}, \tag{7.1}$$

здесь

$$\Gamma(z) = \int_0^{\infty} t^{z-1}e^{-t}dt.$$

Возьмем равноотстоящие узлы на интервале $[0, 7]$

Таблица 7.1

x	0	1	2	3	4	5	6	7
y	0.0000	0.4400	0.5767	0.3391	-0.0660	-0.3276	-0.2767	-0.0040

Дифференциирование самой функции Бесселя представляет собой достаточно сложную задачу. Поэтому построим интерполяционный полином и продифференциируем его. Интерполяционный полином второй степени, проходящий через точки (1,0.44), (2,0.5767), (3,0.3391) представим в виде (см. Главу 6):

$$P_2(x) = -0.071 + 0.6982x - 0.1872x^2. \tag{7.2}$$

Найдем значение производной этого полинома в точке x=2. $J_1'(2) \approx P_2'(2) = -0.0506$.

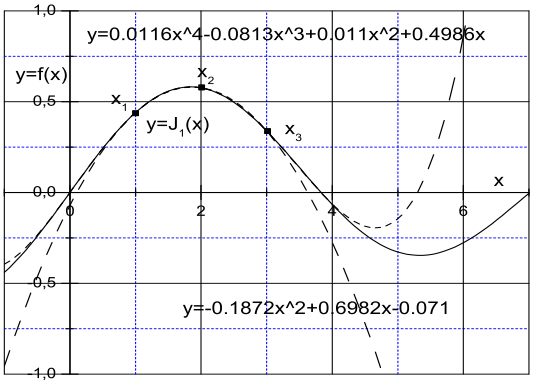

Рис. 7.1:

Интерполяционный полином, построенный по пяти точкам дает лучшее приближение.

$$P_4(x) = 0.4986x + 0.011x^2 - 0.0813x^3 + 0.0116x^4. \tag{7.3}$$

Значение производной в точке 2.0: $J_1'(2) \approx P_4'(2) = -0.0618$. Истинное значение производной $J_1'(2) = -0.0645$. На рисунке 7.1 представлены графики функций $J_1(x)$ уравнение (7.1) – сплошная линия, уравнение (7.2) – пунктирная и уравнение (7.3) – штрихпунктирная линия.

7.2. Конечные разности

Напомним определение производной функции $f(x)$:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{7.4}$$

Определим конечную разность d_k:

$$d_k = \frac{f(x+h_k) - f(x)}{h_k}, \quad k = 1, 2, \ldots, n \tag{7.5}$$

Построим несколько конечных разностей для $f(x) = e^x$ в точке x=1, для нескольких значений h = 0.1, 0.01, 0.001, 0.0001.

$$d_k = \frac{e^{(1+h_k)} - e^1}{h_k}, \quad k = 1, 2, \ldots, n, \; h_k = 10^{-k}. \tag{7.6}$$

Таблица 7.2

Шаг	Разность	Ошибка
0.1	2.8588	-0,140519
0.01	2.7319	-0,013619
0.001	2.7196	-0,001319
0.0001	2.7184	-0,000119

Формула (7.5) определяет правую разность. Аналогичные результаты можно получить используя левую разность. Если использовать центральную разность можно получить более точные результаты. Найдем значение производной функции cos(x) точке $x = 0.3$. Производная $f'(x) = cos'(x) = -sin(x)$. Точное значение sin(0.3) = 0.2955202, соответственно $f'(x) = -sin(x) = -0.2955202$. В таблице 7.4 приведены значения шага и результатов, полученных с помощью правой разности, центральной разности, а также ошибки между точным значением и правой разностью, а также между точным значением и центральной разностью.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{7.7}$$

Таблица 7.3

Шаг	Правая разность	Центральная разность	Ошибка 1	Ошибка 2
0.1	-0.342755	-0.2950279	0.047235	-0.000492
0.01	-0.300292	-0.295514	0.004771	-0.000006
0.001	-0.295971	-0.295533	0.000451	0.000013
0.0001	-0.295384	-0.295433	-0.000136	-0.000087

Как можно видеть из таблицы 7.3 точность расчетов сначала увеличивается, а затем начинает ухудшаться. Это связано со сложным характером зависимости ошибок. В данном случае существуют две ошибки: ошибки потери точности и ошибки округления (см. Главу 2). Ошибки, связанные с потерей точности возникают в тех случаях, когда мы берем разность двух почти равных величин. Ошибки округления возникают из-за неточного представления вещественных чисел в компьютере. В практических расчетах рекомендуется применять следующий критерий: если $|d_{n+1} - d_n| \geq |d_n - d_{n-1}|$, то расчеты следует прекратить. Таким образом для вычислений, проводимых по формулам центральной разности, наилучшее значение получается для значений шага $h = 0.01$.

Упражнения Пусть $f(x) = \sin(x)$, x измеряется в радианах.

1. Вычислите приближение к функции $f'(x) = \sin(0.8)$, используя формулу правой разности. Значения шагов: $h = 0.1,\ h = 0.01,\ h = 0.001$.

2. Сравните полученное приближение со значением $f'(0.8) = \cos(0.8)$.

3. Вычислите приближение к функции для 1 пункта используя формулу центральной разности.

4. Дальность полета снаряда приведена в таблице.

t	S(t)
8.0	17.453
9.0	21.460
10.0	25.752
11.0	30.301
12.0	35.084

   Найдите скорость полета снаряда к 10 секунде.

Формулы численного дифференциирования. Формулы центрированной разности, порядок точности $O(h^2)$:

$$f'(x_0) \approx \frac{f_1 - f_{-1}}{2h},$$
$$f''(x_0) \approx \frac{f_1 - 2f_0 + f_{-1}}{h^2},$$
$$f^{(3)}(x_0) \approx \frac{f_2 - f_1 + 2f_{-1} - f_{-2}}{2h^3},$$
$$f^{(4)}(x_0) \approx \frac{f_2 - 4f_1 + 6f_0 - 4f_{-1} - f_{-2}}{h^4}.$$

Формулы центрированной разности, порядок точности $O(h^4)$:

$$f'(x_0) \approx \frac{-f_2 + 8f_1 - 8f_{-1} + f_{-2}}{12h},$$
$$f''(x_0) \approx \frac{-f_2 + 16f_1 - 30f_0 + 16f_{-1} - f_{-2}}{12h^2},$$
$$f^{(3)}(x_0) \approx \frac{-f_3 + 8f_2 - 13f_1 + 13f_{-1} - 8f_{-2} + f_{-3}}{8h^3},$$
$$f^{(4)}(x_0) \approx \frac{-f_3 + 12f_2 - 39f_1 + 56f_0 - 39f_{-1} + 12f_{-2} - f_{-3}}{6h^4}.$$

7.2.1. Дифференциирование полинома Лагранжа

Для получения формул численного дифференциирования очень удобно пользоваться интерполяционными полиномами Лагранжа и Ньютона. Формулы для равноотстоящих абсцисс, которые лежат справа и слева от точки x_0 можно получить дифференциированием интерполяционного полинома Лагранжа. Приведем несколько формул для правых и левых разностей:

$$f'(x_0) \approx \frac{-3f_0 + 4f_1 - f_2}{2h} \qquad \text{правая разность},$$
$$f'(x_0) \approx \frac{3f_0 - 4f_1 + f_{-2}}{2h} \qquad \text{левая разность},$$
$$f''(x_0) \approx \frac{2f_0 - 5f_1 + 4f_2 - f_2}{h^2} \qquad \text{правая разность},$$
$$f''(x_0) \approx \frac{2f_0 - 5f_1 + 4f_{-2} - f_{-3}}{h^2} \qquad \text{левая разность}$$
$$f^{(3)}(x_0) \approx \frac{-5f_0 + 18f_1 - 24f_2 + 14f_3 - 3f_4}{2h^3}$$
$$f^{(3)}(x_0) \approx \frac{5f_0 - 18f_{-1} + 24f_{-2} - 14f_{-3} + 3f_{-4}}{2h^3}$$

Покажем, как получить формулу $f'(x_0) \approx \frac{-3f_0+4f_1-f_2}{2h}$. Построим интерполяционнй полином Лагранжа по трем точкам $x_0,\ x_1,\ x_2$:

$$f(t) \approx f_0\frac{(t-x_1)(t-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{(t-x_0)(t-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{(t-x_0)(t-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Продифференциируем по t:

$$f(t) \approx f_0\frac{2t-(x_1+x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{2t-(x_0+x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{2t-(x_0+x_1)}{(x_2-x_0)(x_2-x_1)}$$

Так как мы используем равноотстоящие узлы $\Rightarrow x_i - x_j = (i-j)h$, и так как $t = x_0$:

$$f(t) \approx f_0 \frac{-3h}{2h^2} + f_1 \frac{2h}{h^2} - f_2 \frac{h}{2h^2}$$

Приводя к общему знаменателю получаем:

$$f(t) \approx f_0 \frac{-3}{2h} + f_1 \frac{4}{2h} - f_2 \frac{1}{2h}$$

Что и требовалось получить.

Точность вычисления производных. Для равноотстоящих узлов погрешность в вычислении правой разности для функции $f'(x)$ не превышает $\frac{h^3}{3} f'''(\xi)$

Упражнение: Получить формулы дифференциирования полинома Лагранжа для $f''(x_0)$ и $f^{(3)}(x_0)$.

7.2.2. Дифференциирование полинома Ньютона

Аналогично (7.4) можно определить производные для интерполяционного полинома Ньютона. Построим полином Ньютона второго порядка, который приближает функцию $f(t)$ в узлах $t_0,\ t_1,\ t_2$:

$$P(t) = a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1), \tag{7.8}$$

где $a_0 = f(t_0),\ a_1 = (f(t_1) - f(t_0))/(t_1 - t_0)$, и

$$a_2 = \frac{\frac{f(t_2)-f(t_1)}{t_2-t_1} - \frac{f(t_1)-f(t_0)}{t_1-t_0}}{t_2 - t_0}.$$

Производная от (7.8) будет равна:

$$P'(t) \quad = \quad a_1 + a_2((t - t_0) + (t - t_1))$$

Вычислим производную в точке $t = t_0$:

$$P'(t_0) = a_1 + a_2(t_0 - t_1) \approx f'(t_0). \tag{7.9}$$

Если выбрать различные значения узлов мы получим правые и левые разности второго и более высокого порядков. Если выбрать $t_0 = x,\ t_1 = x + h,\ t_2 = x - h$, получим выражения для

$$a_1 = \frac{f(x+h) - f(x)}{h},$$
$$a_2 = \frac{f(x+h) - 2f(x) + f(x-h)}{2h^2}$$

Подставляя эти значения в (7.9) получим:

$$P'(x) = \frac{f(x+h) - f(x)}{h} + \frac{-f(x+h) + 2f(x) - f(x-h)}{2h}$$

получим отюда:

$$P'(x) = \frac{f(x+h) - f(x-h)}{2h} \approx f'(x) \tag{7.10}$$

Это центрированная разность для полинома Ньютона.

Пусть теперь $t_0 = x,\ t_1 = x+h,\ t_2 = x+2h$, тогда

$$a_1 = \frac{f(x+h) - f(x)}{h},$$
$$a_2 = \frac{f(x) - 2f(x+h) + f(x+2h)}{2h^2}.$$

Подставив эти значения в уравнение (7.9) получим формулу правой разности для полинома Ньютона:

$$P'(x) = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \approx f'(x) \tag{7.11}$$

Аналогично можно получить и другие формулы для дифференцирования полинома Ньютона.

Упражнение: Пусть $f(x) = \sin(x)$.

- Найти $f''(0.5)$ по формулам дифференциирования полинома Лагранжа, с шагом $h = 0.05$.
- Найти $f''(0.5)$ по формулам дифференциирования полинома Лагранжа, с шагом $h = 0.01$.
- Найти $f''(0.5)$ по формулам дифференциирования полинома Лагранжа, с шагом $h = 0.1$.

Какой из результатов более точный?

Пример 7.1 Пусть $f(x) = x^3$. Найти приближение $f'(1)$ по (7.11) для шага $h = 0.05$. Вычислим

$$P'(x) = (-3 * 1^3 + 4 * 1.05^3 - 1.1^3)/0.1 = 2.995.$$

Точное значение производной x^3 в точке $x = 1$ равно: $f'(1) = 3 \cdot 1^2 = 3$. Таким образом разница между точным значением и найденным с помощью дифференциирования полинома Ньютона равна 0.005.

7.3. Численное интегрирование

Задачи численного интегрирования возникают в различных областях науки и техники. Часто вычислить определенный интеграл аналитически очень трудно. В таких случаях используют численное вычисление определенного интеграла:

$$I(x) = \int_a^b f(x)dx \tag{7.12}$$

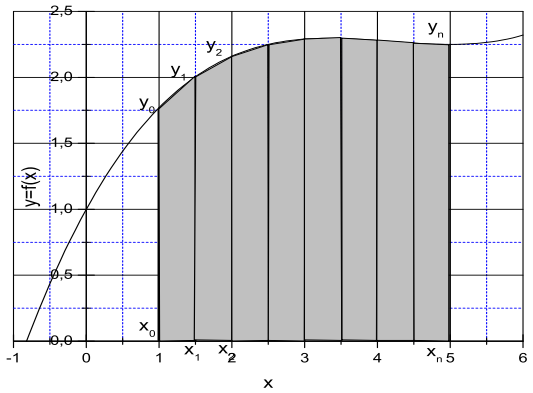

Рис. 7.2:

На рисунке 7.2 изображена функция $f(x)$. Определенный интеграл (7.12) представляет собой площадь фигуры, на отрезке $[a, b]$.

Одним из способов вычисления определенного интеграла является нахождение такой функции $\varphi(x)$, которая приближает функцию $f(x)$ таким образом, что новую функцию $\varphi(x) \approx f(x)$ вычислить легче, чем $f(x)$. Часто, в качестве такой функции выбирают интерполяционный полином Лагранжа $L_n(x)$. Тогда

$$f(x) = L_n(x) + E_n(x), \tag{7.13}$$

здесь $E_n(x)$ остаточный член. Подставив (7.13) в (7.12) получим:

$$I = \int_a^b L_n(x)dx + \int_a^b E_n(x)dx$$

Численное нахождение определенного интеграла на отрезке $[a, b]$ заменим вычислением функции $f(x)$ в конечном числе выбранных узлов

$x_0,\ x_1,\ x_2,\ \ldots,\ x_n$. Узлы можно выбрать как равноотстоящими, так и неравноотстоящими. Как известно (см. **Глава 6.**) существует единственный полином $L_n(x)$ степени $\leq n$, который проходит через n+1 точку $x_0,\ x_1,\ x_2,\ \ldots,\ x_n$. Этот полином можно использовать для замены функции $f(x)$ на отрезке $[a,b]$. Получающиеся формулы называются квадратурными формулами Ньютона – Котеса. Предположим, что $x_k = x_0 + kh$ равноотстоящие узлы, и $f_k = f(x_k)$.

Приведем первые четыре формулы Ньютона – Котеса:

$$\int_{x_0}^{x_1} f(x)dx \approx \frac{h}{2}(f_0 + f_1), \qquad \text{формула трапеций,} \tag{7.14}$$

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}(f_0 + 4f_1 + f_2), \qquad \text{формула Симпсона,} \tag{7.15}$$

$$\int_{x_0}^{x_3} f(x)dx \approx \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3), \qquad \text{формула Симпсона } \frac{3}{8}, \tag{7.16}$$

$$\int_{x_0}^{x_4} f(x)dx \approx \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4), \quad \text{формула Буля.} \tag{7.17}$$

Рассмотрим точность формул Ньютона – Котеса (7.14) – (7.17). Более подробно точность данных методов приведена в параграфе "Оценка погрешности". Пусть $f(x)$ имеет необходимое количество производных. Тогда ошибка $E_n(x)$ для формулы трапеций (7.14) имеет порядок точности $O(h^2)$. Если $f \in C^2[a,b]$ тогда:

$$\int_{x_0}^{x_1} f(x)dx \cong \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12}f^{(2)}(\xi), \qquad \xi \in [a,b]. \tag{7.18}$$

Формула Симпсона имеет порядок точности $O(h^3)$. Пусть $f \in C^4[a,b]$, тогда:

$$\int_{x_0}^{x_2} f(x)dx \cong \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f^{(4)}(\xi), \qquad \xi \in [a,b]. \tag{7.19}$$

Формула Симпсона $\frac{3}{8}$ имеет также третий порядок точности. Пусть $f \in C^4[a,b]$, тогда:

$$\int_{x_0}^{x_3} f(x)dx \cong \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80}f^{(4)}(\xi), \qquad \xi \in [a,b]. \tag{7.20}$$

Формула Буля имеет пятый порядок точности $O(h^5)$. Пусть $f \in C^6[a,b]$, тогда:

$$\int_{x_0}^{x_4} f(x)dx \cong \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945}f^{(6)}(\xi), \qquad \xi \in [a,b]. \tag{7.21}$$

Формулы Ньютона – Котеса получаются из интерполяционного полинома Лагранжа. Заменим функцию $f(x)$ интерполяционным полиномом Лагранжа:

$$\int_{x_0}^{x_n} f(x)dx \approx \int_{x_0}^{x_n} L_n(x)dx = \int_{x_0}^{x_n} (\sum_{k=0}^{n} f_k L_{n,k}(x))dx = \tag{7.22}$$
$$\sum_{k=0}^{n} (\int_{x_0}^{x_n} f_k L_{n,k}dx) = \sum_{k=0}^{n} (\int_{x_0}^{x_n} L_{n,k}dx) f_k = \sum_{k=0}^{n} \omega_k f_k.$$

В качестве примера построим формулу Симпсона. Интерполяционный полином Лагранжа в этом случае записывается в виде:

$$L_2(x) = f_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \tag{7.23}$$
$$+ f_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Значения функций в (7.23) являются постоянными, отсюда следует:

$$\int_{x_0}^{x_2} f(x)dx \approx f_0 \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1 \int_{x_0}^{x_2} \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \tag{7.24}$$
$$+ f_2 \int_{x_0}^{x_2} \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Введем новые переменные: $x = x_0 + ht,\ \ dx = hdt$. В этих новых переменных узлы будут $x_k = x_0 + kh \Rightarrow (x_i - x_j) = (i-j)h$ и $(x - x_i) = h(t-i)$. Пределы интегрирования станут равными $t = 0$ и $t = 2$ вместо x_0 и x_2. Подставляя новые переменные в (7.24), получим:

$$\int_{x_0}^{x_2} f(x)dx \approx f_0 \int_0^2 \frac{h(t-1)h(t-2)}{(-h)(-2h)}hdt + f_1 \int_0^2 \frac{h(t-0)h(t-2)}{(h)(-h)}hdt +$$
$$+ f_2 \int_0^2 \frac{h(t-0)h(t-1)}{(2h)(h)}hdt = f_0\frac{h}{2}\int_0^2 (t^2-3t+2)dt -$$
$$- f_1 h \int_0^2 (t^2-2t)dt + f_2\frac{h}{2}\int_0^2 (t^2-t)dt = \qquad (7.25)$$
$$f_0\frac{h}{2}(\frac{t^3}{3}-\frac{3t^2}{2}+2t)\Big|_0^2 - f_1 h(\frac{t^3}{3}-t^2)\Big|_0^2 + f_2\frac{h}{2}(\frac{t^3}{3}-\frac{t^2}{2})\Big|_0^2 =$$
$$= \frac{h}{3}(f_0+4f_1+f_2).$$

Аналогично можно получить и остальные формулы Ньютона – Котеса.

Формулы Ньютона – Котеса для n точек получаются применением n раз выведенных выше формул на $4 \cdot n$ интервалах. Таким образом, получим:

$$I(h) \cong \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \ldots + f_n) \qquad \text{формула трапеций} \qquad (7.26)$$

$$I(h) \cong \frac{h}{3}(f_0 + 2f_{2i-1} + 4f_{2i} + \ldots + f_n) \quad \text{формула Симпсона} \qquad (7.27)$$

$$I(h) \cong \frac{3h}{8}(f_0 + 3f_{2i-1} + 3f_{2i} + \ldots + f_n) \quad \text{формула Симпсона 3/8} \qquad (7.28)$$

$$I(h) \cong \frac{2h}{45}\sum_{k=1}^{n}(7f_{4k-4} + 32f_{4k-3} + 12f_{4k-2} + 32f_{4k-1} + 7f_{4k}) \text{ формула Буля} \qquad (7.29)$$

Пример 7.2 Рассмотрим функцию $f(x) = xe^x$ на отрезке [0,2]. Аналитически вычисленное значение интеграла равно: $I(h) = 8.3891$.
Выберем равноотстоящие узлы $x_0 = 0.0$, $x_1 = 0.5$, $x_2 = 1.0$, $x_3 = 1.5$, $x_4 = 2.0$. Значения функции в этих точках равны $f(x_0) = 0.0$, $f(x_1) = 0.8244$, $f(x_2) = 2.7183$, $f(x_3) = 6.7225$, $f(x_4) = 14.77811$. Используем формулы Ньютона – Котеса для вычисления интегралов. Интеграл, вычисленный по формуле трапеций (7.26) дает результат $I(h) = 8.584469$. На рисунке 7.3 изображен график функции $f(x) = xe^x$ на отрезке [0,2]. Начальные значения для этого варианта приведены в файле integ.dat:

```
0.0       2.0       0.33333333
1
```

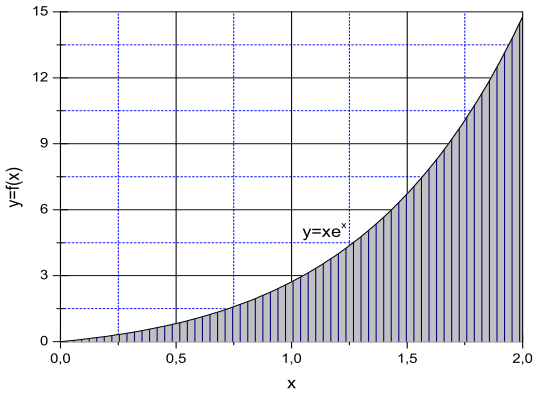

Рис. 7.3:

Ниже приведена программа, которая вычисляет определенные интегралы методами трапеций, Симпсона и Симпсона 3/8. Выбор метода осуществляется с помощью параметра key = 1 – метод трапеций, key = 2 метод Симпсона, key = 3 метод Симпсона 3/8. Функция задается в подпрограмме f. Начальные данные считываются из файла с именем integ.dat. Он состоит из двух строк – в первой задаются значения a, b и h, а во второй – key. Для функции

$$y = 1.0 + \exp(-x)cos(4.0x) \tag{7.30}$$

на отрезке [0,1], с шагом h=1.0/3.0 значение интеграла $I = 1.314397$, рисунок 7.4. Истинное значение интеграла равно 1.308250604626...

Ниже приведен файл integ.dat для функции (7.30):

```
0.0       1.0       0.33333333
3
```

Программа 7.1 Программа вычисления определенного интеграла

```
    program integral
      real::a, b, Integ,num
      integer key
      external f
!  key = 1 метод трапеций, key = 2 метод Симпсона,
```

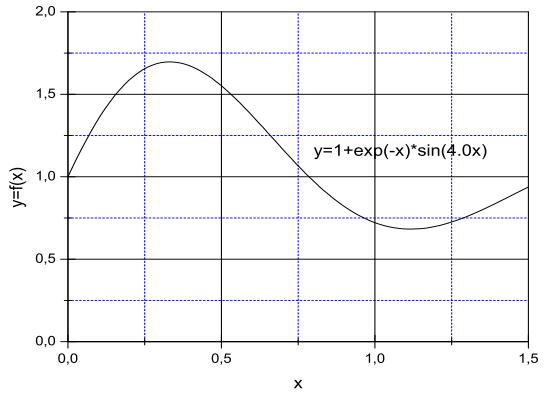

Рис. 7.4:

```
!  key = 3 метод Симпсона 3/8
      open(7,file='integ.dat')
!  Чтение из файла начала и конца отрезков интегрирования и шага
      read(7,91) a,b,h
!  Чтение из файла метода решения
      read(7,93) key
  91  format(3f10.0)
  93  format(i3)
      num = (b-a)/h+1
     if( key == 1 ) then
       call trap(a,b,h,num,Integ,f)
     elseif( key == 2 ) then
       call Simpson(a,b,h,num,Integ,f)
     elseif( key == 3 ) then
       call Simpson3_8(a,b,h,num,Integ,f)
     else
       write(*,*) 'You are missing'
       stop
     endif
     write(*,*) ' Integ=', Integ
    end program integral

    function f(x)
      real f,x
```

```
  f = 1.0+exp(-x)*cos(4.*x)
end function f

subroutine Trap(a,b,h,num,Integ,f)
  real a,b,Integ,f,num
  s = (f(a)-f(b))/2.
 do i = 1,num-1
   s = s + f(a+i*h)
 enddo
 Integ = s*h
end subroutine Trap

subroutine Simpson(a,b,h,num,Integ,f)
   real a,b,Integ,num
   h = (b-a)/(2.0*num)
   s = f(a)/2.
   x = a
  do i = 1,num
    x = x + h
    s = s + 2.0*f(x)
    x = x+h
    x1 = f(x)
    s = s+x1
  enddo
  Integ = (2*s+x1)*h/3.
end subroutine Simpson

subroutine Simpson3_8(a,b,h,num,Integ,f)
   real a,b,Integ,num
   x = a
  do i = 2,num-1
    x = x + h
    s = s + f(x)
  enddo
  Integ = (f(a)+3.0*s+f(b))*3*h/8.
end subroutine Simpson3_8
```

7.3.1. Кратные интегралы

Необходимость численного вычисления многократных интегралов часто встречается в инженерных, физических задачах, при проведении математического моделирования. Рассмотрим для примера вычисление двукратного интеграла в прямоугольной области $\Re\{(x,y):\ a \leqslant x \leqslant b,\ c \leqslant y \leqslant d\}$.

$$I = \int_a^b \int_c^d f(x,y)dxdy. \tag{7.31}$$

Объем полученной фигуры можно вычислить разными методами. Рассмотрим метод, аналогичный методу Симпсона. В этом методе мы разбиваем искомый объем на:

$$I = \int_a^b \int_c^d f(x,y)dxdy \approx S \cdot f\Big(\frac{a+b}{2}, \frac{c+d}{2}\Big), \tag{7.32}$$

здесь $\triangle x = (b-a)/n$, $\triangle y = (d-c)/m$, $S = \triangle x \cdot \triangle y$, n и m - количество отрезков разбиения по каждой стороне соответственно. Вычисление двукратного интеграла можно свести в этом случае к:

$$I = \int_a^b \int_c^d f(x,y)dxdy \approx \sum_i S_i \cdot f(x_i, y_i).$$

Здесь S_i площадь i прямоугольника $x_i, y_i \in S_i$ - внутренняя точка i прямоугольника.

Представим двойной интеграл (7.31) в виде:

$$I = \int_a^b \int_c^d f(x,y)dxdy = \int_a^b dx \int_c^d f(x,y)dy.$$

Внешний интеграл по формуле Симпсона можно вычислить по формуле (7.18):

$$I = \frac{\triangle x}{3}\Big[\int_c^d f(a,y)dy + 4\int_c^d f\big(\frac{a+b}{2}, y\big)dy + \int_c^d f(b,y)dy\Big]. \tag{7.33}$$

Если применить (7.18) к каждому из интегралов (7.33) - получим:

$$I_1 = \int_c^d f(a,y)dy \approx \frac{\triangle y}{3}\big[f(a,c) + 4f(a, \frac{c+d}{2}) + f(a,d)\big],$$

$$I_2 = \int_c^d f(\frac{a+b}{2}, y)dy \approx \frac{\triangle y}{3}\big[f\big(\frac{a+b}{2}, c\big) + 4f\big(\frac{a+b}{2}, \frac{c+d}{2}\big) + f(\frac{a+2}{2}, d)\big],$$

$$I_3 = \int_c^d f(b, y)dy \approx \frac{\triangle y}{3}\big[f(b, c) + 4f\big(b, \frac{c+d}{2}\big) + f(b, d)\big].$$

Если воспользоваться формулами для численного вычисления каждого из этих интегралов окончательно получим:

$$I \approx \frac{\triangle x \triangle y}{9}\Big[f(a,c) + f(a,d) + f(b,c) + f(b,d) + \tag{7.34}$$

$$2\big[f(a, \frac{c+b}{2} + f(b, \frac{c+d}{2}) + f(\frac{a+b}{2}, c + f(\frac{a+b}{2}, d) + \tag{7.35}$$

$$+4f(\frac{a+b}{2}), \frac{c+d}{2})\big]\Big]. \tag{7.36}$$

7.3.2. Оценка погрешности

Оценим погрешность метода трапеций. На каждом шаге вычисления определенного интеграла методом трапеций мы совершаем некоторую ошибку. Ее можно оценить следующим образом - воспользуемся формулой для оценки остаточного члена интерполяционного полинома первого порядка:

$$\zeta_N = \Big|\int_{x_n}^{x_{n+1}} R_N(x)dx\Big| \leqslant \int_{x_n}^{x_{n+1}} \frac{\max|f^{(N+1)}(\xi)|}{(N+1)!}\Delta x + \max\Big|\prod_{n=0}^{N}(x - x_n)\Big|.$$

Для формулы трапеций получим:

$$\zeta_N = \int_{x_n}^{x_{n+1}} \frac{\max|f''(\xi)|}{2}\max|(x - x_n)(x - x_{n+1})|dx = \frac{\max|f''(\xi)|}{12}\triangle x^3.$$

На отрезке $[a, b]$ общая погрешность метода трапеций примет вид:

$$\zeta_N \leqslant \sum_{n=1}^{N}|\zeta_m| \leqslant \frac{\max|f''(\xi)|}{12}\triangle x^3 \cdot N = \frac{\max|f''(\xi)|}{12}\triangle x^2 \cdot (b-a). \tag{7.37}$$

Если провести аналогичные рассуждения для других методов, можно получить погрешности

- для формулы прямоугольников (со средней точкой):

$$\zeta_N \leqslant \frac{b-a}{24} \max |f''_x(x)| \triangle x^2. \tag{7.38}$$

- для метода Симпсона:

$$\zeta_N \leqslant \frac{b-a}{180} \max |f^{(4)}_x(x)| \triangle x^4. \tag{7.39}$$

- для метода Симпсона 3/8:

$$\zeta_N \leqslant \frac{b-a}{2880} \max |f^{(4)}_x(x)| \triangle x^4. \tag{7.40}$$

Если подинтегральная функция $f(x)$ для формулы Симпсона не имеет четвертой производной, то погрешность возрастает в несколько раз:

$$\zeta_N \leqslant \frac{b-a}{12} \max |f'''_x(x)| \triangle x^3.$$

7.4. Упражнения

1. Проинтегрируйте интерполяционный полином Лагранжа

$$L_1(x) = f_0 \frac{(x-x_1)}{(x_0-x_1)} + f_1 \frac{(x-x_0)}{(x_1-x_0)}$$

на интервале $[x_0, x_1]$ и выведите формулу трапеций.

2. Проинтегрировать функции $f(x)$ на интервале $[0, 1]$. Использовать все четыре формулы (7.26) – (7.29). Шаг $h = 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$.

- $f(x) = \sin(\pi x)$
- $f(x) = \sin(\sqrt{x})$
- $f(x) = 1 + e^{-x}\cos(4x)$

3. Методом трапеций найти значение интеграла $I(h) = \int_0^1 \sin(x)dx = 0.4597$ для значений шага h=0.5, 0.25, 0.1.

4. Методом Симпсона найти значение интеграла $I(h) = \int_0^5 \sin(x^2)dx = 0.52791728$ для значений шага h=0.02, 0.01, 0.005.

5. Методом Буля найти значение интеграла $I(h) = \int_0^1 (1 + e^{-x}\sin(4x))dx = 1.3082506$ для значений шага h=1.0, 0.5, 0.1, 0.05.

6. Методом Симпсона 3/8 найти значение интеграла $I(h) = \int_0^3 \frac{\sin(2x)}{(1+x^2)}dx = 0.67175786$ для значений шага h=0.5, 0.05, 0.005.

7. Методами трапеций, Симпсона, Симпсона 3/8 и Буля найти значение интеграла $I(h) = \int_0^2 \frac{1}{(x^2+1/10)}dx = 4.47139939$ для значений шага h=0.5, 0.05, 0.005.

Глава 8

Численные методы решения обыкновенных дифференциальных уравнений

8.1. Введение

Обыкновенным дифференциальным уравнением (ОДУ) называется уравнение вида:

$$f(x, y, y', y'', \ldots, y^{(n)}) = 0 \tag{8.1}$$

в которое входит независимая переменная x, искомая функция $y(x)$, и ее производные до n – го порядка включительно. Порядок старшей производной определяет порядок обыкновенного дифференциального уравнения. Общий вид обыкновенного дифференциального уравнения:

$$a_0 y + a_1 y' + a_2 y'' + \ldots + a_{n-1} y^{(n-1)} + a_n y^{(n)} + \psi(x) = 0.$$

Определение 8.1 *Если $\psi(x) \equiv 0$ – такое уравнение называется однородным, в противном случае – неоднородным. Если все a_i константы или зависят только от x, то такое уравнение называется линейным, в противном случае – нелинейным.*

Определение 8.2 *Функция*

$$\Phi(x, y, C_1, C_2, \ldots, C_n) \tag{8.2}$$

называется общим интегралом уравнения (8.1).

Уравнение (8.2) связывает искомую функцию $y(x)$, независимую переменную x и n постоянных интегрирования.

Определение 8.3 *Функция*

$$f(x) = \varphi(x, y, C_1, C_2, \dots, C_n) \tag{8.3}$$

называется общим решением обыкновенного дифференциального уравнения (8.1).

Таким образом в общий интеграл искомая функция входит неявно, а в общем решении искомая функция определяется явно.

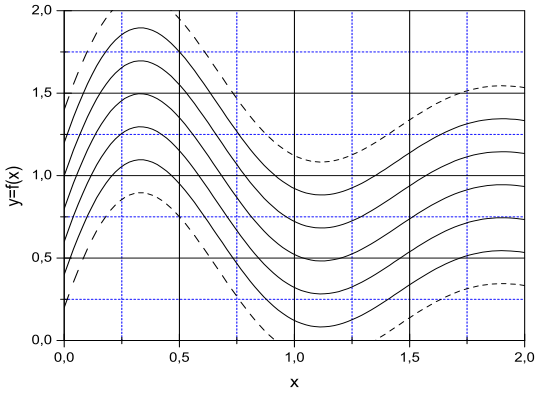

Рис. 8.1:

На рисунок 8.1 приведено общее решение обыкновенного дифференциального уравнения.

Пример 8.1 Рассмотрим обыкновенное дифференциальное уравнение

$$y'(x) = 1 - e^x \tag{8.4}$$

Общим решением этого уравнение является функция

$$y(x) = x - e^x + C \tag{8.5}$$

На плоскости x, y функция (8.5) представляет собой бесконечный набор кривых, которые отличаются друг от друга на постоянную C. Для определения этой константы нам необходимо задать дополнительные условия.

В случае уравнения n порядка необходимо задать n дополнительных условий. Если подставить эти условия в уравнение (8.3) и решить полученную систему относительно постоянных интегрирования $(C_1, C_2, \ldots, C_n)$ и подставить полученные решения в (8.2), то мы получим частный интеграл $\Psi_1(x, y(x)) = 0$.

Определение 8.4 *Если все дополнительные условия задаются в одной точке x_0, то совокупность ОДУ и этих дополнительных условий называется* **задачей Коши**. *Дополнительные условия называются начальными данными.*

Если в задаче (8.4) поставить условие (задачу Коши): при $x_0 = 0 \;\; y_0 = -1$ то из (8.5) найдем значение C=0. Уравнение $y(x) = x - e^x$ является частным решением уравнения (8.4) и определяет одну интегральную кривую.

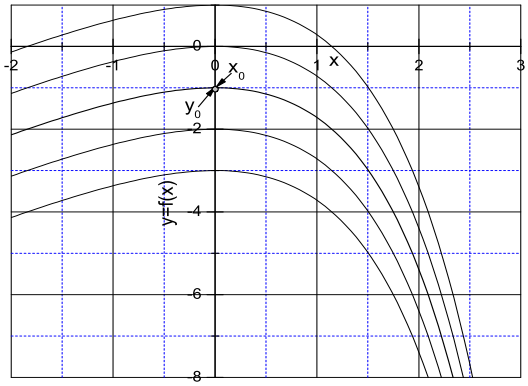

Рис. 8.2:

На рисунок 8.2 приведено частное решение обыкновенного дифференциального уравнения (8.4). Точка $x_0 = 0 \;\; y_0 = -1$ позволяет выбрать одну кривую, которая является частным решением.

Таким образом: пусть на отрезке [a,b] задана функция $y' = f(x, y)$ и $y(x_0) = x_0$. Тогда решением задачи Коши является такая функция $y = y(x)$, которая на заданном отрезке удовлетворяет условиям:

$$y'(x) = f(x, y(x)), \quad y(x_0) = y_0, \quad \forall x \in [a, b]$$

Определение 8.5 *Если дополнительные условия задаются более чем в одной точке, то такая задача называется краевой задачей, а эти*

дополнительные условия называются граничными или краевыми условиями.

Задача Коши является корректно поставленной, если заданы начальные условия и функция $f(x, y(x))$ непрерывна на отрезке [a,b]. В этом случае система (8.1) имеет решение, но в общем случае не единственное. Определим условие Липшица:

$$|f(x, y) - f(x, y_0)| \leq L|y - y_0|, \tag{8.6}$$

где L константа.

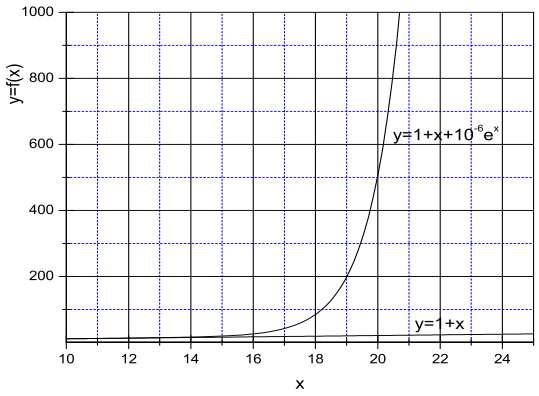

Рис. 8.3:

Теперь, если у нас есть корректно поставленная задача Коши и, кроме того, выполняется условие Липшица то решение такой задачи существует и единственно. Но существование и единственность решения не гарантирует численного решения задачи ОДУ.

Существуют обыкновенные дифференциальные уравнения, которые удовлетворяют всем необходимым условиям, но в то же время найти численное решение таких задач невозможно. Такие задачи называют плохо обусловленными.

Приведем пример плохой обусловленности. Пусть

$$y'(x) = y - x, \quad y(0) = 1, \quad \Rightarrow y(x, C) = 1 + x + Ce^x \Rightarrow C = 0. \tag{8.7}$$

Вычислим значение функции в точке 100: $y(100) = 100$. Изменим начальное условие. Пусть теперь $y(0) = 1.000001$. В этом случае $C = 10^{-6}$

и решение примет вид: $y(100) = 2.7 \cdot 10^{37}$. Таким образом, изменив начальные условия на 10^{-6} мы получили решение, которое отличается на ~ 35 порядков.

На рисунке 8.3 приведено решение уравнения (8.7) в диапазоне значений x от 10 до 35.

Обыкновенное дифференциальное уравнение n – го порядка путем замены переменных можно свести к системе n уравнений первого порядка.

$$\begin{cases} y' = z_1(x) \\ z_1' = z_2(x) \\ \dots \\ z_{n-2}' = z_{n-1}(x) \\ z_{n-1}' = f(x, y, z_1(x), \dots z_{n-1}(x)) \end{cases}$$

Для того, чтобы поставить задачу Коши необходимо задать n дополнительных условий в точке x_0:

$$\begin{cases} y(x_0) = y_0 \\ z_1(x_0) = y_1 \\ \dots \\ z_{n-1}(x_0) = y_{n-1} \end{cases}$$

8.2. Метод Эйлера

Получить аналитическое решение задачи Коши удается не всегда. Поэтому существуют другие методы решения обыкновенных дифференциальных уравнений. Наиболее часто используемыми являются численные методы. Использование конечных разностей позволяет свести обыкновенное дифференциальное уравнение (или систему уравнений обыкновенных дифференциальных уравнений) к системе алгебраических уравнений. Такой переход осуществляется следующим образом.

a. На отрезке интегрирования вводится разностная сетка. Разностная сетка представляет собой конечный дискретный набор узлов. На границе этого набора узлов задаются граничные условия.

b. Осуществляется переход от непрерывных дифференциальных уравнений к системе алгебраических уравнений. Специальным образом записываются сеточные уравнения в граничных узлах, в зависимости от типа используемых граничных условий.

c. Полученную систему алгебраических уравнений можно решить одним из методов, приведенных в выше.

8.2.1. Граничные условия

В граничных узлах разностной сетки задаются граничные условия. Эти условия могут быть трех типов. Пусть обыкновенное дифференциальное уравнение задано на отрезке [0,1]:

$$y' = f(x, y) \quad x \in [0, 1]. \tag{8.8}$$

1. **Граничное условие Дирихле.** На одной из границ – $x = 0$ или $x = 1$ (или на обоих границах) задаются значения искомой функции, т.е.

$$\begin{cases} y(0) = y_0, \\ y(1) = y_1. \end{cases} \tag{8.9}$$

   где y_0 и y_1 константы

2. **Граничное условие Неймана.** На границах $x = 0$ и (или) $x = 1$ задаются производные искомой функции, т.е.

$$\begin{cases} y'(0) = y_0, \\ y'(1) = y_1. \end{cases} \tag{8.10}$$

3. **Граничное условие Робина.** На границах $x = 0$ и (или) $x = 1$ задаются комбинации искомой функции и ее производной, т.е.

$$\begin{cases} \alpha_0 y'(0) + \beta_0 y(0) = y_0, \\ \alpha_1 y'(1) + \beta_1 y(1) = y_1. \end{cases} \tag{8.11}$$

Граничные условия (8.9), (8.10) и (8.11) называются еще граничными условиями первого, второго и третьего рода.

Замена дифференциального уравнения алгебраическими уравнениями может осуществлятся различными способами. Одним из самых распространенных и простых численных методов решения ОДУ является метод Эйлера. Основной идеей метода Эйлера является разложение искомой функции в ряд Тейлора в окрестности некоторой точки x_0.

Пусть для обыкновенного дифференциального уравнения $y' = f(x, y)$ существует корректно поставленная задача Коши на отрезке [a,b], то

есть $y(a) = y_0$. Введем на отрезке [a,b] систему узлов x_k. Для удобства выберем длины интервалов $x_{k+1} - x_k$ одинаковыми, $h = \frac{(b-a)}{n}$, где n число точек разбиения. Используя теорему Тейлора разложим функцию $f(x)$ в окрестности точки $x_0 = a$, предполагая, что функции $y(x),\ y'(x),\ y''(x)$ непрерывны:

$$y(x) = y(x_0) + y'(x_0)(x - x_0) + \frac{1}{2!}y''(x_0)(x - x_0)^2 + \frac{1}{3!}y'''(x_0)(x - x_0)^3 + \dots ,$$

обозначив $h = x - x_0$, $\xi \in [x, x_0]$ и ограничившись первыми членами разложения получим:

$$y(x_1) = y(x_0) + hf(x_0) + y''(\xi)\frac{h^2}{2}.$$

Если выбрать $h \ll 1$, то членами второго порядка можно пренебречь, тогда

$$y_{k+1} = y_k + h \cdot f(x_k). \tag{8.12}$$

Формула (8.12) реализует метод Эйлера. Графически, решение задачи Коши методом Эйлера представляет собой ломаную линию, которая в каждом узле совпадает с касательной в этой точке.

Пример 8.2 Решить уравнение со следующими начальными условиями:

$$y' = x^2 - y, \quad y(x) = Ce^{-x} + x^2 - 2x + 2, \quad y(0) = 0 \Rightarrow C = -2.$$

$$y(x) = -2e^{-x} + x^2 - 2x + 2, \tag{8.13}$$

На рисунке 8.4 приведен график функции $y(x) = -2e^{-x} + x^2 - 2x + 2$.

Программа 8.1 Программа метода Эйлера

```
program Euler
  implicit none
  integer,parameter::n=100
  integer::i,j
  real::y(n),yt(n),x,h,f,f1,z(2,n),e(n)
  f(x)   = -2*exp(-x)+x*x-2.*x+2
  f1(x,y) = x*x-y
   x=0.
   h=0.1
```

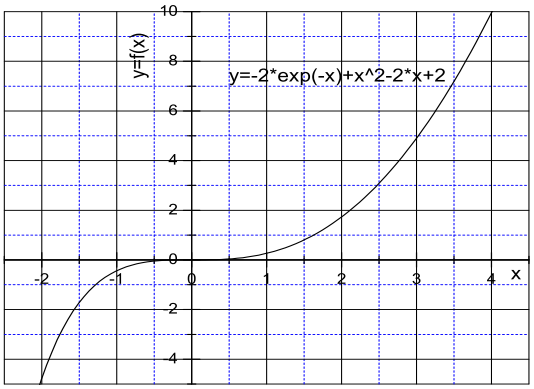

Рис. 8.4:

```
      y(1)=0.

     do i =2,n-1
!Численное решение методом Эйлера
       y(i+1) = y(i)+h*f1(x,y(i))
!Точное решение
       yt(i+1) = f(x)
       x=x+h
     enddo
      z(1,:) = y
      z(2,:) = yt
      e=abs(y-yt)
      write(*,91) (z(1,i),z(2,i),e(i),i=1,n)
 91   format(2x,3(f12.6,2x))
    end program Euler
```

В программах Эйлера и Эйлера–Коши (смотри программу 8.2), для вычисления значений функции и аналитического решения, используются оператор–функции f(x) и f1(x,y).

Упражнения Методом Эйлера найти решение задачи Коши для следующих дифференциальных уравнений:

1. $y' = 2xy^2$, точное решение $y(x) = \frac{1}{C-x^2}$, $y(0) = 1$.

2. $y' = -xy$, $y(x) = Ce^{-x^2/2}$, $y(0) = 1$.

3. $y' = e^{-2x} - 2y, \quad y(x) = Ce^{-2x} + xe^{-2x}, \quad y(0) = 1.$
4. $y' = 3x + 3y, \quad y(x) = Ce^{-3x} + x - \frac{1}{3}, \quad y(0) = 1.$
5. $y' = x^2 - y, \quad y(x) = Ce^{-x} + x^2 - 2x + 2, \quad y(0) = 0.$
6. $y' = 1 + y^2, \quad y(x) = \tan(x + \frac{\pi}{4}), \quad y(0) = 1.$
7. $y' = y^2, \quad y(x) = \frac{1}{C-x}, \quad y(0) = 1.$

8.2.2. Точность метода Эйлера

При выводе уравнения метода Эйлера мы пренебрегали членами порядка $y''(\xi)\frac{h^2}{2}$. Если это единственная ошибка, то к концу вычислений на отрезке [a,b] после n шагов накопленная ошибка будет равна:

$$\sum_{k=1}^{n} y''(\xi)\frac{h^2}{2} \approx ny''(\xi)\frac{h^2}{2} = \frac{hn}{2}y''(\xi)h = \frac{(b-a)y''(\xi)}{2}h = O(h)$$

Пример 8.3 Решить методом Эйлера интегродифференциальное уравнение:

$$y' = 1.3y - 0.25y^2 - 0.0001y\int_0^x y(\xi)d\xi,$$

шаг h=0.2, y(0)=250, на отрезке [0,20]. Для вычисления определенного интеграла воспользуемся формулой трапеций. Из (8.12) получим:

$$y_{k+1} = y_k + h(1.3y_k - 0.25y_k^2 - 0.0001y_k\int_0^x y(\xi)d\xi)$$

Если заменить $\int_0^x y(\xi)d\xi$ на соответствующую сумму по формуле трапеций (7.14), то

$$y_{k+1} = y_k + h(1.3y_k - 0.25y_k^2 - 0.0001y_k I_k(h))$$

где $I_0(h) = 0$ и $I_k(h) = I_{k-1}(h) + h/2 \cdot (y_{k-1} + y_k)$.

8.3. Модифицированный метод Эйлера

Метод Эйлера имеет низкую точность и используется в основном в учебных целях. Его редко применяют при проведении научных расчетов. Для определения значения на новом шаге в методе Эйлера используется один

шаг. Существуют более сложные двух и более шаговые методы. В таких методах, чтобы получить значение на новом шаге нужно использовать два или более шагов. Рассмотрим двухшаговую модификацию метода Эйлера схему предиктор – корректор. На первом шаге вычисляется предварительное значение на новом шаге. Этот шаг называется предиктор. На втором шаге, который называется корректор, вычисленные на первом шаге значения корректируются и находится окончательное значение искомой функции.

Одним из таких методов является метод Эйлера – Коши. Будем искать решение задачи Коши:

$$y'(x) = f(x, y(x)), \quad x \in [a, b], \quad y(x_0) = y_0$$

Если применить фундаментальную теорему математического анализа и проинтегрировать функцию $y'(x)$ на отрезке $[x_k, x_{k+1}]$:

$$\int_{x_k}^{x_{k+1}} f(x, y(x, y(x))dx = \int_{x_k}^{x_{k+1}} y'(x)dx = y(x_{k+1}) - y(x_k). \tag{8.14}$$

Если решить уравнение (8.14) относительно $y(x_k)$ получим:

$$y(x_{k+1}) = y(x_k) + \int_{x_k}^{x_{k+1}} f(x, y(x, y(x))dx \tag{8.15}$$

Для вычисления интеграла в (8.15) воспользуемся формулой прямоугольников со средней точкой:

$$y(x_{k+\frac{1}{2}}) \approx y(x_k) + hf(x_k + \frac{h}{2}, y(x_{k+1})) \tag{8.16}$$

здесь $h = (x_{k+1} - x_k)$.

Значение $f(x_k + \frac{h}{2}, y(x_{k+1}))$ неизвестно. Это означает, что (8.16) является, в общем случае, нелинейным относительно $y(x_{k+1})$. Такое уравнение можно решить, используя методы, изложенные в Главе 4, в частности метод простой итерации (4.8). Тогда первое приближение – шаг предиктор можно получить, используя метод Эйлера:

$$\widetilde{y}_{k+\frac{1}{2}} = y_k + \frac{h}{2} \cdot f(x_k, y_k), \quad x_{k+\frac{1}{2}} = x_k + h/2. \tag{8.17}$$

Второй шаг – корректор и окончательное значение определяется по формуле прямоугольников со средней точкой:

$$y_{k+1} = y_k + h \cdot (f(x_k, y_k) + f(x_{k+\frac{1}{2}}, \widetilde{y}_{k+\frac{1}{2}})). \tag{8.18}$$

Точность метода Эйлера – Коши можно определить используя оценку остаточного члена в формуле прямоугольников. Он равен

$$\zeta_N \leqslant y''(\xi_k)\frac{h^3}{24}. \tag{8.19}$$

Если это единственная ошибка, то после n шагов накопленная ошибка метода Эйлера – Коши будет равна:

$$\sum_{k=1}^{n} y''(\xi_k)\frac{h^3}{24} \approx \frac{b-a}{24}y''(\xi)h^2 = O(h^2).$$

Таким образом метод Эйлера – Коши обладает вторым порядком точности.

Пример 8.4 Решить модифицированным методом Эйлера – Коши уравнение

$$y' = x^2 - y, \quad y(x) = -2e^{-x} + x^2 - 2x + 2.$$

Программа 8.2 Программа метода Эйлера – Коши

```
   program EulerCauchy
     implicit none
!  Variables
     integer,parameter::n=100
     integer::i,j
     real::y(n),ye(n),yz(n),yg(n),yt(n),x(n),h, &
           f,f1,z(4,n),e(3,n)
     f(x)   = -2*exp(-x)+x*x-2.*x+2
     f1(x,y) = x*x-y
!   Body of EulerCauchy
     h =0.1
     y(1)= 0.

     do i = 1,n
       x(i)=(i-1)*h
     enddo

     do i = 2,n-1
        yz(i+1) = yz(i)+h*f1(x(i),yz(i))
     enddo
      ye=yz
```

```
    do i = 2,n-1
       y(i+1) = y(i)+0.5*h*(f1(x(i),y(i))+ &
                f1(x(i+1),yz(i)))
       yg(i+1)= y(i)+0.5*h*(f1(x(i),y(i))+ &
                f1(x(i+1),y(i)+f1(x(i),y(i))))
       yt(i+1) = f(x(i))
    enddo
     z(1,:) = y
     z(2,:) = yz
     z(3,:) = yt
     z(4,:) = yg
     e(1,:) = y-yt
     e(2,:) = yz-yt
     e(3,:) = yg-yt
     write(*,91) (z(1,i),z(2,i),e(1,i),e(2,i),e(3,i),i=1,n)
91   format(2x,5(f12.6,2x))
   end program EulerCauchy
```

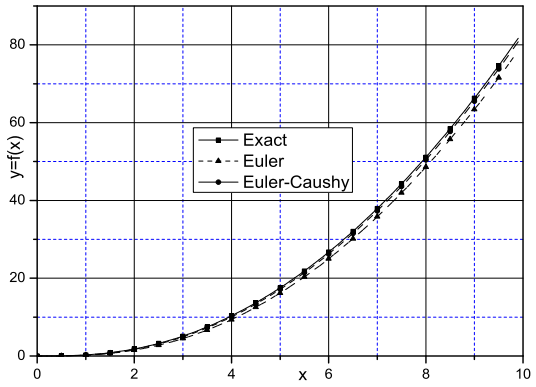

Рис. 8.5:

На рисунке 8.5 приведены графики функций, полученных по точному решению (уравнение (8.13)), на рисунке обозначены прямоугольниками, решение по методу Эйлера – на рисунке обозначены треугольниками и по методу Эйлера – Коши на рисунке обозначены окружностями. Отметим более точное совпадение точного решения и решения, полученного по методу Эйлера – Коши.

Упражнения. Решить методом Эйлера и методом Эйлера – Коши следующие дифференциальные уравнения:

1. $y' = -xy,\ \ \mathrm{y}(0) = 1,\ \ y(x) = e^{-\frac{x^2}{2}}$,
2. $y' = 3y + 3x,\ \ \mathrm{y}(0) = 1,\ \ y(x) = 4/3e^{3x} - x - 1/3$,
3. $y' = 2xy^2,\ \ \mathrm{y}(0) = 1,\ \ y(x) = \frac{1}{1-x^2}$,
4. $y' = e^{-2x} - 2y,\ \ \mathrm{y}(0) = 1/10,\ \ y(x) = 1/10e^{-2x} + xe^{-2x}$,
5. $y' = x^2 - 2y,\ \ \mathrm{y}(0) = 1,\ \ y(x) = 3/4e^{-2x} + 1/2x^2 - 1/2x + 1/4$.

8.3.1. Метод Гюна

Другим подходом в решении дифференциальных уравнений является интегрирования уравнения

$$y'(x) = f(x, y) \tag{8.20}$$

вдоль $y(x)$. Для получения решения в некоторой точке $\xi \in [a, b]$ проинтегрируем функцию (8.20) на интервале $[x_i, x_{i+1}]$.

$$\int_{x_i}^{x_{i+1}} f(x, y(x))dx = \int_{x_i}^{x_{i+1}} y'(x)dx = y(x_{i+1}) - y(x_i). \tag{8.21}$$

Если решить уравнение (8.21) относительно $y(x_{i+1})$ получим:

$$y(x_{i+1}) = y(x_i) + \int_{x_i}^{x_{i+1}} f(x, y(x))dx. \tag{8.22}$$

Интеграл в уравнении (8.22) можно вычислить используя численные методы. Воспользуемся методом трапеций (7.14), тогда из (8.22) получим

$$y(x_{i+1}) \approx y(x_i) + \frac{h}{2}(f(x_i, y(x_i)) + f(x_{i+1}, y(x_{i+1}))). \tag{8.23}$$

В уравнении (8.23) нам неизвестно значение $y(x_{i+1})$. Это неизвестное значение можно вычислить, воспользовавшись методом Эйлера (8.12). Если подставить полученное значение в (8.23) получим:

$$y(x_{i+1}) = y(x_i) + \frac{h}{2}(f(x_i, y(x_i)) + f(x_i, y_i + hf(x_i, y_i))). \tag{8.24}$$

Полученное уравнение определяет метод Гюна. Для решения этим методом уравнения на отрезке $[a,b]$ необходимо найти первое приближение методом Эйлера, а затем уточнить, используя метод трапеций.

$$\begin{aligned} \widetilde{y}_{i+1} &= y_i + h\,f(x_i, y_i), \\ x_{i+1} &= x_i + h, \\ y_{i+1} &= y_i + \frac{h}{2}(f(x_i, y(x_i)) + f(x_{i+1}, \widetilde{y}_{i+1}). \end{aligned} \tag{8.25}$$

Ошибка метода Гюна состоит из остаточного члена в методе трапеций

$$-y''(\xi_k)\frac{h^3}{12}. \tag{8.26}$$

Если это единственная ошибка, то после N шагов обшая ошибка не будет превышать

$$\sum_{k=1}^{N} y''(\xi_k)\frac{h^3}{12} \approx \frac{b-a}{12}y''(\xi)h^2 = O(h^2). \tag{8.27}$$

Пример 8.5 Решить дифференциальное уравнение

$$y' = 0.2(x - y) \tag{8.28}$$

методами Эйлера и Гюна. Точное решение имеет вид:

$$y = 3e^{-0.5x} + x - 2. \tag{8.29}$$

На рисунке 8.6 приведены графики решения уравнения (8.28) методом Эйлера – штрихпунктирая линия, методом Гюна – пунктирная линия и точное решение – сплошная линия.

Программа 8.4 Программа метода Гюна

```
program Gune
  implicit none

  integer,parameter::nx=30
  integer::i
  real,dimension(nx)::x,y,y1,y2,yt,z(3,nx),err(2,nx)
  real::h=0.1,f,fn,u,u1,u2,yy
```

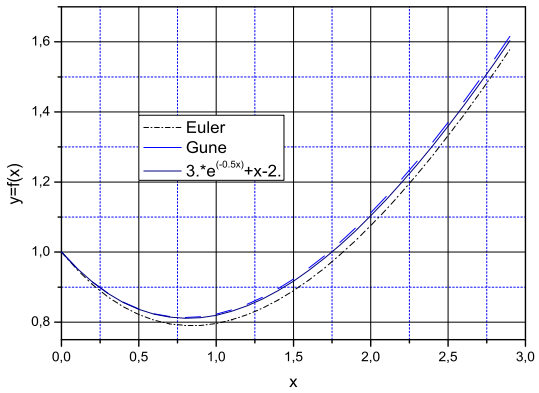

Рис. 8.6:

```
do i = 1,nx
  x(i) = (i-1)*h
enddo
y(1)=1.
y1(1)=1.
y2(1)=1.
yt(1)=1.
yy = 1.

do i = 1,nx-1
  yy = yy+h*f(x(i),yy)
  y(i+1) = yy
enddo
  y1 = y

do i = 1,nx-1
   y(i+1) = y(i)+h/2.*(f(x(i),y(i))+f(x(i+1),y1(i+1)))
enddo

do i =2,nx
  yt(i) = fn(x(i),y(i))
enddo
z(1,:) = y1
z(2,:) = y
```

```
      z(3,:) = yt
      err(1,:) = y1-yt
      err(2,:) = y-yt
      open(7,file='Gune.dat')
       write(7,91) (x(i),z(1,i),z(2,i),z(3,i),i=1,nx)
      close(7)
      open(8,file='Error.dat')
       write(8,92) (x(i),err(1,i),err(2,i),i=1,nx)
      close(8)
 91   format(4(2x,f11.6))
 92   format(3(2x,f11.6))
    end program Gune

    real function f(x,y)
      real x,y
      f = 0.5*(x-y)
    end function f

    real function fn(x,y)
      real x,y
      fn = 3.*exp(-0.5*x)-2.+x
    end function fn
```

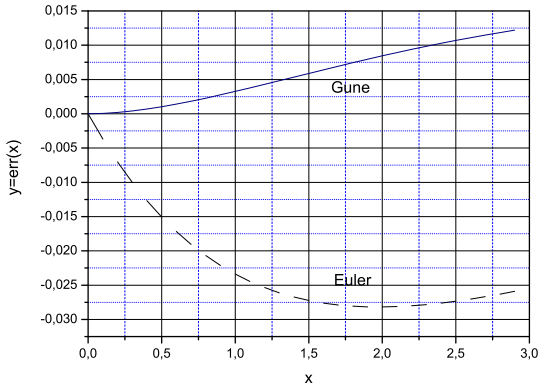

Рис. 8.7:

На рисунке 8.7 приведены графики ошибок для уравнения (8.28), полученные методом Эйлера и методом Гюна. На этом рисунке пунктирной

линий обозначены разность между решением по методу Эйлера и точным решением, сплошной линией разность между решением по методу Гюна и точным решением.

8.4. Методы Рунге – Кутта

Для того, чтобы получить разностные схемы более высокого порядка, используются методы Рунге – Кутта. Применение методов более высокого порядка позволяет добиться нужной точности не проводя большого количества дополнительных вычислений. Так как методы, подобные методу Эйлера, имеют маленькую точность, для достижения нужной точности необходимо использовать малый шаг h. При этом количество вычислений становится неоправданно большим. Это часто приводит к накоплению ошибок округления.

В основе методов высокого порядка лежит идея оставить в разложении Тейлора члены высокого порядка. При этом вычислять производные высоких порядков нужно не дифференциированием функций, а вычислением искомой функции в некоторых дополнительных точках на отрезке $[x_k, x_{k+1}]$.

Методы Рунге – Кутта позволяют добиться точности $O(h^n)$, при этом можно выбрать достаточно большие значения n. Одним из самых распространенных, из-за своей экономичности и точности, являются методы Рунге – Кутта второго и более высоких порядков точности. В этих методах на каждом шаге требуется вычислить дополнительно несколько промежуточных значений k_i.

Рассмотрим метод Рунге – Кутта четвертого порядка точности. На каждом шаге интегрирования необходимо дополнительно вычислить четыре промежуточные значения.

$$\begin{aligned} k_1 &= hf(x_k, y_k), \\ k_2 &= hf(x_k + h/2, y_k + k_1/2), \\ k_3 &= hf(x_k + h/2, y_k + k_2/2), \\ k_4 &= hf(x_k + h, y_k + k_3). \end{aligned} \tag{8.30}$$

Окончательное значение y на новом шаге вычисляется по формуле:

$$y_{k+1} = \frac{1}{6}(f_1 + 2f_2 + 2f_3 + f_4). \tag{8.31}$$

Остаточный член в формуле (8.31) равен:

$$E = -y^{(4)}(\xi)\frac{h^5}{2880}. \tag{8.32}$$

Если на каждом шаге ошибка не превосходит (8.32), то после n шагов общая ошибка будет равна:

$$\sum_{k=1}^{n}(y^{(4)}(\xi_k)\frac{h^5}{2880} \approx \frac{b-a}{5760}y^{(4)}(\xi)h^4 \approx O(h^4). \tag{8.33}$$

Уменьшение шага в два раза для методов четвертого порядка точности приводит к уменьшению окончательной ошибки в 16 раз. Покажем это на следующем примере.

Пример 8.6 Решим уравнение $y' = x^2 - 2y$, на отрезке $[0, 2]$, с начальным значением $y(0) = 1$. Сравним решения, полученные при различных длинах шага с точным решением $y(x) = 3/4e^{-2x} + 1/2x^2 - 1/2x + 1/4$. В приведенной ниже программе метода Рунге – Кутта четвертого порядка функция $f(x, y)$ задается в отдельной подпрограмме *real function* $f(x, y)$, точное решение в подпрограмме – функции *real function* $ft(x)$. Шаг $h = 0.1$. Численное решение совпадает с точным решением до четвертого знака после запятой.

Программа 8.3 Программа метода Рунге – Кутта

```
program Runge_Cutta
  integer,parameter::nx=11
   real(4)::x(11),yt(nx),yrk(xn)
   real(4)::k0,k1,k2,k3,ie,iem,irk
   h = 0.1
   x(1)  = 1.
   yt(1) = 1.
  yrk(1) = 1.

   do i = 1,nx
     x(i) = (i-1)*h
   enddo

   do i = 2,nx
     yt(i) = ft(x(i))
   enddo
   write(*,91)
```

```
    do i = 2,11
      k0 = h*f(x(i-1),yrk(i-1))
      k1 = h*f((x(i-1)+0.5*h),(yrk(i-1)+0.5*k0))
      k2 = h*f((x(i-1)+0.5*h),(yrk(i-1)+0.5*k1))
      k3 = h*f((x(i-1)+h),(yrk(i-1)+k2))
      yrk(i) = yrk(i-1)+(k0+2.*k1+2.*k2+k3)/6.
      irk = abs(yrk(i)-yt(i))*100./yt(i)
      write(*,93) x(i),yrk(i),yt(i)
    enddo
91   format(t7,' x ',t20,' yrk ',t33,' yt ')
93   format(5(2x,f11.7))
95   format(10x,3(1x,'(',f15.7,')'))
  end

  real function f(x,y)
    real x,y
    f = x**2-2.*y
  end function f

  real function ft(x)
    real x
    ft = 3./4.*exp(-2*x)+0.5*x**2-0.5*x+0.25
  end function ft
```

Результаты расчетов по методу Рунгу – Кутта и точное решение.

x	метод РК	точное решение
0.0000000	1.0000000	1.0000000
0.1000000	0.8190508	0.8190480
0.2000000	0.6727448	0.6727400
0.3000000	0.5566147	0.5566087
0.4000000	0.4670035	0.4669967
0.5000000	0.4009168	0.4009096
0.6000000	0.3559031	0.3558956
0.7000000	0.3299553	0.3299477
0.8000000	0.3214298	0.3214224
0.9000000	0.3289815	0.3289742
1.0000000	0.3515086	0.3515015
1.1000000	0.3881093	0.3881024
1.2000000	0.4380452	0.4380385

```
1.3000001    0.5007117    0.5007052
1.4000000    0.5756139    0.5756075
1.5000000    0.6623465    0.6623403
1.6000000    0.7605776    0.7605717
1.7000000    0.8700358    0.8700300
1.8000001    0.9904985    0.9904929
1.9000000    1.1217836    1.1217780
2.0000000    1.2637421    1.2637367
```

8.4.1. Модификация метода Рунге – Кутта

Приведенные выше методы решения обыкновенных дифференциальных уравнений имели постоянный шаг интегрирования h. Это обстоятельство накладывает определенные ограничения на вычисления с заданной точностью, так как заранее не известна оптимальная длина шага. Одним из хорошо известных методов, в котором величина шага интегрирования вычисляется в процессе расчета, является метод Рунге – Кутта – Фехлберга (РКФ45).

В этом методе используется алгоритм, который позволяет определить оптимальную длину шага. В алгоритме вычисляются два очередных значения функции, которые потом сравниваются между собой. Из этого сравнения и определяется необходимая величина шага. Метод РКФ45 состоит из вычисления шести промежуточных значений $k_i,\ i = 1, \ldots, 6$:

$$
\begin{aligned}
k_1 &= hf(x_k, y_k),\\
k_2 &= hf(x_k + h/4, y_k + k_1/4),\\
k_3 &= hf(x_k + 3/8h, y_k + 3/32k_1 + 9/32k_2),\\
k_4 &= hf(x_k + 12/13h, y_k + 1932/2197k_1 - 7200/2197k_2 + 7296/2197k_3)\\
k_5 &= hf(x_k + h, y_k + 439/216k_1 - 8k_2 + 3680/513k_3 - 854/4104k_4),\\
k_6 &= hf(x_k + h/2, y_k - 84/27k_1 + 2k_2 - 3544/2565k_3 + 1859/4104k_4 -\\
&\quad -11/40k_5).
\end{aligned}
$$

Окончательное значение на новом шаге получается из:

$$y_{k+1} = y_k + 25/216k_1 + 1408/2565k_3 + 2197/4101k_4 - k_5/5. \qquad (8.34)$$

Кроме того вычисляется еще одно значение на $k+1$ шаге:

$$\widetilde{y}_{k+1} = y_k + 16/135k_1 + 6656/12.825k_3 + 28.561/56.430k_4 - 9/50k_5 + 2/55k_6. \qquad (8.35)$$

В уравнениях (8.34) и (8.35) параметр k_2 не участвует.

Из сравнения этих двух значений можно получить скалярный множитель λ, на который нужно умножить шаг h для того, чтобы достичь требуемой точности. Пусть нам необходимо проводить вычисления с точностью ε, тогда:

$$\lambda = \left(\frac{\varepsilon h}{2|\widetilde{y}_{k+1} - y_{k+1}|} \right)^{\frac{1}{4}} \approx 0.84 \left(\frac{\varepsilon h}{2|\widetilde{y}_{k+1} - y_{k+1}|} \right)^{\frac{1}{4}}$$

Умножая длину очередного шага h на λ мы получим оптимальный шаг для достижения требуемой точности ε.

Пример 8.7 Рассмотрим задачу Коши для уравнения $y = x^2 - y$, на отрезке $[0, 1]$. Точное решение определяется уравнением $y = -e^{-x} + x^2 - 2x + 2$. Зададим начальный шаг $h = 0.1$ и точность расчета $\varepsilon = 10^{-5}$. Ниже приведена таблица, в котороq в первом столбце приведено значение x, во втором вычисленное по методу RKF45, в третьем точное решение и в четвертом очередное значение шага расчета h.

x	RKF45	Точное	Шаг h
0.1000000	1.0000000	1.0000000	0.1000000
0.1000000	0.9051626	0.9051626	0.0900000
0.1900000	0.8291085	0.8291408	0.0810000
0.2710000	0.7687687	0.7688245	0.0729000
0.3439000	0.7213947	0.7214674	0.0656100
0.4095100	0.6846185	0.6847029	0.0590490
0.4685590	0.6564338	0.6565260	0.0531441
0.5217031	0.6351619	0.6352590	0.0478297
0.5695328	0.6194124	0.6195123	0.0430467
0.6125795	0.6080424	0.6081435	0.0387420
0.6513215	0.6001192	0.6002203	0.0348678
0.6861894	0.5948857	0.5949861	0.0313810
0.7175704	0.5917313	0.5918301	0.0282429
0.7458134	0.5901654	0.5902625	0.0254186
0.7712320	0.5897967	0.5898918	0.0305024
0.8017344	0.5906667	0.5907589	0.0274521
0.8291865	0.5926839	0.5927731	0.0247069
0.8538934	0.5955070	0.5955931	0.0222362
0.8761296	0.5988696	0.5989525	0.0200126
0.8961422	0.6025654	0.6026453	0.0180113
0.9141536	0.6064369	0.6065139	0.0162102

Нужно отметить, что для достижения требуемой точности, шаг расчета в программе автоматически уменьшился почти на порядок. В программе метода RKF45, как и в предыдущей программе, исходная функция и точное решение вычисляются в отдельных подпрограммах – функциях. Некоторые массивы например z(2,nx) ввыедены для визуализации расчетов.

Программа 8.4 Программа метода Рунге – Кутта – Фехлберга

```
    program RKF45
      integer,parameter::nx=20
      real::k1,k2,k3,k4,k5,k6,lambda,y1,yy,y2,ye,h
      real::z(2,nx),f,y,eps = 1.e-05,h1(nx),x1(nx)
      h=0.1
      h1 = h
      x = 0.
      y1 = 1.
      y2 = 1.
      ye = 1.

     do i = 1,20
       k1 = h*f(x,y1)
       k2 = h*f(x+h*0.25,y1+k1*0.25)
       k3 = h*f(x+3.*h/8.,y1+3./32*k1+9./32*k2)
       k4 = h*f(x+12./13*h,y1+1932./2197.*k1-7200./2197.*k2+ &
            7296./2197.*k3)
       k5 = h*f(x+h,y1+439./216.*k1-8*k2+3680./513.*k3- &
            845./4104.*k4)
       k6 = h*f(x+0.5*h,y1-84./27.*k1+2.*k2-3544./2565.*k3+ &
            1859./4104.*k4-11./40.*k5)
       yy = y1 + 25./216.*k1 + 1408./2565.*k3 + &
            2197./4101.*k4-1./5.*k5
       y2 = y1 + 16./135.*k1 + 6656./12825.*k3 + &
            28561./56430.*k4 - 9./50.*k5 + 2./55.*k6
       lambda = 0.84*((eps*h)/(abs(yy-y2)))**0.25
! Контроль изменения шага по времени
     if( lambda < 0.9 ) lambda=0.9
     if( lambda > 1.2 ) lambda=1.2
       x = x + h
       x1(i) = x
       h = h*lambda
```

```
      ye = ft(x)
      z(1,i) = y2
      z(2,i) = ye
      h1(i) = h
      write(*,91) x,yy,y2,ye,lambda,h
   91 format(2x,6(f10.7,2x))
      y1 = yy
    enddo

      open(7,file='rkf45.dat')
      write(7,93) (x1(i),z(1,i),z(2,i),h1(i),i=1,nx)
   93 format(2x,4(f11.7,2x))
   end program RKF45

   real function f(x,y)
     real x,y
     f = x*x - y
   end function f

   real function ft(x)
     real x
     ft = -exp(-x)+x*x-2.*x+2.
   end function ft
```

В программе введены ограничения на резкие изменения шага по пространству. Без этого ограничения решение может стать неустойчивым.

8.5. Многошаговые методы

Рассмотренные в предыдущих параграфах методы численного решения обыкновенных дифференциальных уравнений – Эйлера, Эйлера – Коши, Рунге – Кутта и другие являются одношаговыми методами. В таких методах для вычисления в следующем $k+1$ узле используется информация, полученная в предыдущем, k узле. Однако, после того, как вычислены значения в некоторых последовательных узлах можно воспользоваться этой информацией для проведения более точных расчетов. Существует ряд методов в которых используется информация из двух, трех, четырех и более уже вычисленных узлах.

8.5.1. Метод Адамса

Рассмотрим, для примера, четырех шаговый метод Адамса, в котором, для вычисления $k+1$ узла используется информация из $k-3$, $k-2$, $k-1$, k узлов. Для начала расчетов по методу Адамса необходимо каким – либо образом вычислить значения в первых четырех узлах. Информация, полученная из четырех предыдущих узлов используется для вычисления корректирующего члена, который повышает точность расчетов, а также для определения длины очередного шага h. Таким образом многошаговые методы часто являются методами с переменной длиной шага.

Метод Адамса выводится из уравнения:

$$y(x_{k+1}) \approx y(x_k) + \int_{x_k}^{x_{k+1}} f(x, y(x))dx \tag{8.36}$$

При выводе этого метода воспользуемся интерполирующим полиномом Лагранжа, который задается в уже вычисленных точках (x_{k-3}, f_{x-3}), (x_{k-2}, f_{x-2}), (x_{k-1}, f_{x-1}), (x_k, f_x). Исходя из этого можно получить предварительное значение:

$$\widetilde{y}(x_{k+1}) = y_k + \frac{h}{24}(-9f_{k-3} + 37f_{k-2} - 59f_{k-1} + 55f_k) \tag{8.37}$$

Иногда формула (8.37) считается окончательной для метода Адамса. Но существует модификация этого метода, которая называется методом Адамса – Бешфорса – Маултона (Adams – Bashforth – Moulton). Она заключается в следующем. После вычисления промежуточного значения $\widetilde{y}(x_{k+1})$ его можно использовать для построения следующего полинома Лагранжа. Построив этот полином и подставив его в (8.36), интегрируя на отрезке $[x_k, x_{k+1}]$ получаем второй шаг – корректор, таким образом окончательное значение на k+1 шаге будет равно:

$$y(x_{k+1}) = y_k + \frac{h}{24}(f_{k-2} - 5f_{k-1} + 19f_k + 9f(x_{k+1}, \widetilde{y}(x_{k+1})). \tag{8.38}$$

При выводе шага корректор мы использовали информацию, полученную на промежуточном шаге $f(x_{k+1}, \widetilde{y}(x_{k+1}))$. Эту информацию можно использовать либо для повышения точности метода, либо для автоматического изменения длины шага.

Для этого найдем ошибки для шагов предиктор и корректор. Для шага предиктор ошибка составляет:

$$y(x_{k+1}) - \widetilde{y}_{k+1} \approx \frac{251}{720} y^{(5)}(\xi_{k+1})h^5. \tag{8.39}$$

Для шага корректор ошибка равна:

$$y(x_{k+1}) - y_{k+1} \approx -\frac{19}{720} y^{(5)}(\eta_{k+1}) h^5. \tag{8.40}$$

Если из (8.39) и (8.40) исключить $y^{(5)}(x)$, и взять достаточно малый шаг, получим:

$$y(x_{k+1}) - y_{k+1} \approx -\frac{19}{720}(y_{k+1} - \widetilde{y}_{k+1}). \tag{8.41}$$

Формулу (8.41) можно использовать для автоматического увеличения или уменьшения шага. Если мы проводим расчеты с одинарной точностью, тогда зададим значение ожидаемой ошибки $\varepsilon = 5 \cdot 10^{-5}$, и пусть $\vartheta = 10^{-5}$. Тогда если выполняется следующее неравенство – мы будем уменьшать шаг h в два раза.

$$\frac{19}{270} \frac{|y_{k+1} - \widetilde{y}_{k+1}|}{|y_{k+1}| + \vartheta} > \varepsilon \quad \Rightarrow h = h/2.$$

Если выполняется неравенство:

$$\frac{19}{270} \frac{|y_{k+1} - \widetilde{y}_{k+1}|}{|y_{k+1}| + \vartheta} < \varepsilon/100 \quad \Rightarrow h = 2h.$$

Значения ε и ϑ можно подобрать таким образом, чтобы автоматически выбирать длину очередного шага, либо использовать их для достижения требуемой точности. Уменьшение длины шага в два раза приводит к необходимости введения дополнительных узлов, которые расположены между $k-2$ и $k-1$ точками, а также вычисления двух новых значений функций в этих узлах. Для получения значений в этих узлах необходимо построить интерполяционный полином Лагранжа четвертой степени, используя новые четыре точки $x_{k-3/2},\ x_{k-1},\ x_{k-1/2},\ x_k$. Интерполяционная формула для получения значений в этих узлах имеет вид:

$$\begin{array}{rcl} f_{k-1/2} & = & \frac{-5f_{k-4}+28f_{k-3}-70f_{k-2}+140f_{k-1}+35f_k}{128} \\ f_{k-3/2} & = & \frac{3f_{k-4}-20f_{k-3}+90f_{k-1}+60f_{k-1}-5f_k}{128} \end{array} \tag{8.42}$$

Если происходит увеличение шага, необходимо просто пропустить 2 узла.

Упражнение Написать программу, реализующую метод Адамса с автоматическим выбором шага.

8.5.2. Метод Милна – Симпсона

В методе Милна – Симпсона, который тоже основан на схеме предиктор – корректор интегрирование осуществляется на отрезке $[x_{k-3}, x_{k+1}]$:

$$y(x_{k+1}) = y(x_{k-3}) + \int_{x_{k-3}}^{x_{k+1}} f(x, y(x))dx. \tag{8.43}$$

Интерполирующий полином Лагранжа в этом методе строится по точкам (x_{k-2}, f_{k-2}), (x_{k-1}, f_{k-1}), (x_k, f_k). Первый шаг – предиктор можно получить интегрируя (8.43):

$$\widetilde{y}(x_{k+1}) = y(x_{k-3}) + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k) \tag{8.44}$$

Для получения второго шага – корректора как и в методе Адамса можно использовать вычисленное значение $\widetilde{y}(x_{k+1})$ уравнение (8.44). Построим интерполяционный полином Лагранжа по точкам (x_{k-1}, f_{k-1}), (x_k, f_k), $(x_{k+1}, f(x_{k+1}, \widetilde{y}_{k+1}))$. Если проинтегрировать полином на отрезке $[x_{k-1}, x_{k+1}]$ можно получить окончательное значение на новом шаге:

$$y(x_{k+1}) = y(x_{k-1}) + \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1}) \tag{8.45}$$

Остаточный член, как и в методе Адамса равен:

$$y(x_{k+1}) - \widetilde{y}(x_{k+1}) \approx \frac{28}{29}(y_{k+1} - \widetilde{y}_{k+1}). \tag{8.46}$$

Если интегрируемая функция достаточно гладкая, то можно ввести корректирующий параметр η_{k+1}:

$$\eta_{k+1} = \widetilde{y}(x_{k+1}) + \frac{28}{29}(y_k - \widetilde{y}_k). \tag{8.47}$$

Если уравнение (8.47) использовать вместо $\widetilde{y}_{k+1}$, то подставляя в (8.45) можно получить:

$$y_{k+1} = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f(x_{k+1}, \eta_{k+1})). \tag{8.48}$$

Окончательно, метод Милна – Симпсона запишется в виде:

$$\begin{array}{lll} \widetilde{y}_{k+1} & = y_{k-3} + 4h/3(2f_{k-2} - f_{k-1} + 2f_k), & \text{предиктор} \\ \eta_{k+1} & = \widetilde{y}_{k+1} + 28/29(y_k - \widetilde{y}_k), & \text{параметр} \\ f_{k+1} & = f(x_{k+1}, \eta_{k+1}), & \\ y_{k+1} & = y_{k-1} + h/3(f_{k-1} + 4f_k + f_{k+1}). & \text{корректор} \end{array} \tag{8.49}$$

Упражнение Написать программу для решения дифференциальных уравнений методом Милна – Симпсона. Сравнить полученное решение с решением, полученным по методу Адамса.

8.5.3. Метод Хемминга

Существует еще один метод высокого порядка точности – метод Хемминга. Он также основан на методе предиктор – корректор. Запишем без вывода шаг предиктор:

$$\widetilde{y}_{k+1} = y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k), \tag{8.50}$$

Шаг корректор можно представить в виде:

$$\widetilde{\widetilde{y}}_{k+1} = \frac{-y_{k-2} + 9y_k}{8} + \frac{3h}{8}(-f_{k-1} + 2f_k + f_{k+1}), \tag{8.51}$$

Окончательное значение очередного шага находится по формуле:

$$y_{k+1} = \widetilde{\widetilde{y}}_{k+1} + \frac{9}{129}(\widetilde{y}_{k+1} - \widetilde{\widetilde{y}}_{k+1}). \tag{8.52}$$

Приведем программу, разработанную для решения дифференциальных уравнений методом Хемминга.

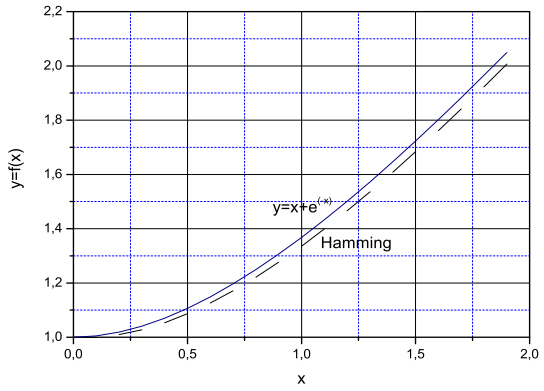

Рис. 8.8:

Программа 8.5 Программа метода Хемминга

```
program Hamming
  integer,parameter::nx=20
  real::x(nx),y(nx),yp(nx),y1(nx),y2(nx),yt(nx),z(3,nx), &
        err(2,nx),dx=0.1
```

```
      y(1)=1.
      yy = 1.
      yt = 1.
      do i = 1,nx
        x(i) = (i-1)*dx
      enddo
      do j = 1,nx-1
         yy = yy + dx*(f(x(j),yy))
         y(j+1) = yy
      enddo
      do i = 4,nx-1
        yy = y(i-3)+4.*dx/3.*(2.*f(x(i-2),y(i-2))- &
             f(x(i-1),y(i-1))+2*f(x(i),y(i)))
        yp(i+1) = yy
      enddo
      do i = 5,nx-1
        y2(i+1) = (-yp(i-2)+9.*y2(i))/8.+3*dx*(-f(x(i-1), &
                  y(i-1))+2.*f(x(i),y(i))+f(x(i+1),y(i+1)))/8.
      enddo
      do i = 1,nx-1
        y1(i+1) = y1(i+1)+9./129.*(y(i+1)-y1(i+1))
      enddo
      do i=2,nx
        yt(i) = fn(x(i))
      enddo
      z(1,:) = x
      z(2,:) = y
      z(3,:) = yt
      err(1,:) = x
      err(2,:) = y-yt
       write(7,91) (z(1,i),z(2,i),z(3,i),i=1,nx)
      close(7)
      open(8,file='Error.dat')
       write(8,92) (err(1,i),err(2,i),i=1,nx)
      close(8)
91    format(3(2x,f11.6))
92    format(2(2x,f11.6))
    end program Hamming
```

```
real function f(x,y)
  real x,y
  f = 1.-exp(-x)
end function f

real function fn(x)
  real x
  fn = x+exp(-x)
end function fn
```

На рисунке 8.8 приведено численное решение, полученное методом Хемминга – пунктирная линия и точное решение – сплошная линия.

8.6. Неявные методы

Определение 8.6 *Разностная схема называется* **явной**, *если правая часть дифференциального уравнения зависит только от уже известных величин.*

Определение 8.7 *Разностная схема называется* **неявной**, *если члены в правой части уравнения зависят от величин на новом шаге.*

Определение 8.8 *Разностная схема называется* **полунеявной**, *если правая часть уравнения зависит как от известных, так и от неизвестных величин.*

$$y_{k+1} = y_k + (\omega f(x_k, y_k) + (1-\omega) f(x_{k+1}, y_{k+1})) \tag{8.53}$$

Здесь ω параметр разностной схемы: $0 \le \omega \le 1$. Если $\omega = 1$ – схема (8.53) явная, если $\omega = 0$ – схема (8.53) неявная, $0 < \omega < 1$ – схема (8.53) полунеявная.

Почти все рассмотренные выше схемы являются явными. Нелинейные уравнения, которые получаются в результате использования неявных схем можно решить одним из изложенных выше (Глава 4) методов: методом деления отрезка пополам, методом золотого сечения, Ньютона, секущих. Будем решать нелинейные уравнения методом простой итерации. Обозначим номер итерации верхним индексом, тогда

$$y_{k+1}^{(i+1)} = y_k + h(\omega f(x_k, y_k) + (1-\omega) f(x_{k+1}, y_{k+1}^{(i)})) \tag{8.54}$$

В качестве начального приближения можно выбрать предыдущее значение y_k, или значение, вычисленное по любому из приведенных выше

методов. Итерационный процесс заканчивается после достижения требуемой точности, т.е. когда разница между предыдущим и текущим приближениями станет меньше ε: $|y^{(i+1)} - y^{(i)}| \leq \varepsilon$.

Пример 8.8 Решить обыкновенное дифференциальное уравнение

$$y' = e^{-2x} - 2y \tag{8.55}$$

неявным методом. Задача Коши: в точке $x_0 = 0$ $y(x_0) = 1$. Точное решение равно:

$$y = e^{-2x}x + e^{-2x} \tag{8.56}$$

Провести сравнение результатов, полученных для различных значений ω в диапазоне $0 \leqslant \omega \leqslant 1$. Ниже приведена программа решения ОДУ неявным методом. Вычисление функции (8.55) осуществляется в подпрограмме *real function* $f(x, y)$, а точное решение (уравнение (8.56)) рассчитывается в *real function* $fn(x, y)$. В массив z(3,nx) записываются значения x, численное решение, найденное неявным методом, и точное решение. Этот массив можно визуализировать с помощью программы Array Visualizer. В массиве err(6,nx) вычисляются разности между численным и точным решениями $|y(x - i) - yt(x_i)|$.

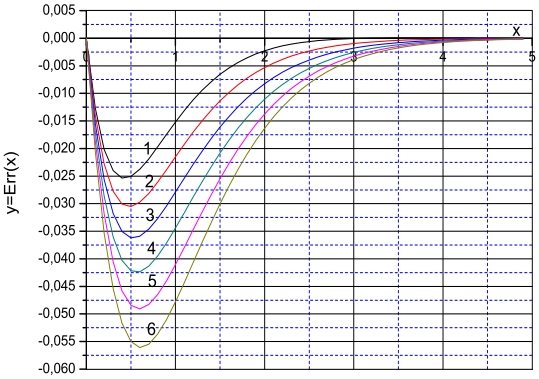

Рис. 8.9:

Программа 8.6 Программа решения ОДУ неявным методом

```
program Implicit
  integer,parameter::nx = 50,maxiter=25
```

```
 real, dimension(nx)::x(nx+1),yt,y1,z(3,nx),err(6,nx)
 real::h, omega, y0, eps=1.0E-05, y2, yy, yold
 h=0.1
 omega = 0.0
 y0 = 1.0
 y1(1) = y0
 yt(1) = y0
 yold = 5.0
 max2 = 0

do i = 1,nx+1
  x(i) = (i-1)*h
enddo

do k = 1,nx
  yt(k) = ft(x(k),y1(k))
enddo

do j = 1,6
  omega = omega + 0.2
  y1(1) = y0
do k = 2,nx
  max1 = 0
  y1(k) = y1(k-1)

 do i = 1,maxiter
   max1 = max1 + 1
   y2 = y1(k)+h*(omega*(f(x(k),y1(k)))+ &
        (1.0-omega)*f(x(k+1),y2))
 if( abs(y2-yold) <= eps ) exit
   yold = y2
 enddo
  if( max1 > max2 ) max2 = max1
   y1(k) = y2
enddo
  err(j,:) = y1-yt
enddo
z(1,:) = x(1:nx)
z(2,:) = y1
```

```
    z(3,:) = yt
    open(7,file='ErrImplicit.dat')
     write(7,91) (x(i),err(1,i),err(2,i),err(3,i), &
     err(4,i),err(5,i),err(6,i),i=1,nx)
    close(7)
91  format(7(1x,f11.6))
  end program Implicit

  real function f(x,y)
    real::x,y
    f = exp(-2.0*x)-2.0*y
  end  function

  real function ft(x,y)
    real:: x,y
    ft = exp(-2.0*x)*(x+1.0)
  end function ft
```

На рисунке 8.9 приведены графики ошибок, полученные по программе 8.6 для уравнения (8.55) и для различных значений ω. Значение ω изменяется в диапазоне от 0 до 1 с шагом $\Delta\omega = 0.2$. Кривая 1 на рисунке отображает значение $\omega = 0$, что соответствует полностью неявной схеме. Кривые 2,3,4,5,6 соответствуют значениям ω равным $\{0.2,\ 0.4,\ 0.6,\ 0.8,\ 1.0\}$ соответственно. Напомним, что значение $\omega = 1$ соответствует явной разностной схеме. Из 8.9 видно, что наименьшую ошибку дает полностью неявная разностная схема, а наибольшую - явная схема. Максимальная ошибка, полученная по полностью неявной схеме в два раза меньше максимальной ошибки, полученной по явной разностной схеме.

8.7. Решение систем ОДУ

Как известно, любое дифференциальное уравнение n-го порядка можно привести к системе n дифференциальных уравнений первого порядка. Часто и сами задачи приводят не к одному обыкновенному дифференциальному уравнению, к системе ОДУ. Рассмотрим например задачу Лоттка – Вольтерра хищник – жертва. Пусть $x(t)$ количество жертв, например зайцев, которых поедают хищники – например лисы. Пусть $y(t)$ – количество хищников. Тогда, если в этой цепочке нет других участников,

изменение по времени численности жертв можно описать уравнением:

$$\frac{dx}{dt} = Ax - Bxy \tag{8.57}$$

Здесь A и B коэффициенты. Изменение числа хищников по временем можно задать в виде:

$$\frac{dy}{dt} = Cy - Dxy \tag{8.58}$$

Поставим задачу Коши. Пусть в момент времени $t = t_0$, $x(t_0) = x_0$, $y(t_0) = y_0$. Пусть в начальный момент времени $t_0 = 80$, а $y_0 = 30$. Необходимо проинтегрировать эту систему уравнений на отрезке $[t_0, t^*]$.

В общем виде систему двух обыкновенных дифференциальных уравнений можно записать в виде:

$$\begin{cases} \frac{dx}{dt} = f(t, x, y), \\ \frac{dy}{dt} = g(t, x, y). \end{cases} \tag{8.59}$$

с начальными данными:

$$\begin{cases} x(t_0) = x_0, \\ y(t_0) = y_0. \end{cases} \tag{8.60}$$

Решением системы ОДУ (8.59) с начальными условиями (8.60) являются две функции $\varphi(t)$ и $\psi(t)$.

Рассмотрим систему дифференциальных уравнений:

$$\begin{cases} \frac{dx}{dt} = x + 2y, \\ \frac{dy}{dt} = 3x + 2y. \end{cases} \tag{8.61}$$

с начальными значениями:

$$\begin{cases} x(0) = 6, \\ y(0) = 4. \end{cases}$$

Аналитическим решением задачи Коши является система уравнений:

$$\begin{cases} x(t) = 4e^{4t} + 2e^{-t}, \\ y(t) = 6e^{4t} - 2e^{-t}. \end{cases} \tag{8.62}$$

Решить численно систему (8.61) на отрезке [a,b] можно, используя рассмотренные выше методы. Решение этой системы методом Эйлера будет выглядеть так:

$$\begin{cases} x_{k+1} = x_k + hf(t_k, x_k, y_k), \\ y_{k+1} = y_k + hf(t_k, x_k, y_k), \end{cases} \tag{8.63}$$

где $t_{k+1} = t_k + h, \quad (k = 0, 1, \ldots, n-1)$.

Пример 8.9 Методом Эйлера решить систему уравнений (8.61), начальными данными (8.62). Сравнить с аналитическим решением (8.62). Ниже представлена программа решения систем обыкновенных дифференциальных уравнений методом Эйлера. Подпрограммы функции *real function* $f(t,x,y)$ и *real function* $g(t,x,y)$ задают исходные уравнения (8.61), а *real function* $ft(t,x,y)$ и *real function* $t(t,x,y)$ аналитические решения.

Программа 8.7 Программа решения систем ОДУ методом Эйлера

```
    program Euler_Systems
!  Решение системы дифференциальных уравнений методом Эйлера
      integer,parameter::nx=20
      real::x1,x2,y1,y2,t=0,h=0.05,z(2,nx),z1(2,nx),zz(5,nx)
      x1 = 6.
      y1 = 4.
      zz(2,1) = x1
      zz(3,1) = x1
      zz(4,1) = y1
      zz(5,1) = y1
     do i = 2,nx
       x1 = x1+h*f(t,x1,y1)
       y1 = y1+h*g(t,x1,y1)
       t = t+h
       zz(1,i) = t
       zz(2,i) = x1
       zz(3,i) = 4.*exp(4.*t)+2.*(-t)
       zz(4,i) = y1
       zz(5,i) = 6.*exp(4*t)-2.*exp(-t)
     enddo
     open(7,file='ES.dat')
     write(7,91) (zz(1,i),zz(2,i),zz(3,i),zz(4,i), &
                  zz(5,i),i=1,nx)
 91  format(5(1x,f11.6))
    end program Euler_Systems

    real function f(t,x,y)
      real x,y
      f = x+2.*y
```

```
end

real function g(t,x,y)
  real x,y
  g = 3.*x+2.*y
end

real function ft(t,x,y)
  real x,y
  ft = 4.*exp(4.*t)+2.*(-t)
end function ft

real function gt(t,x,y)
  real x,y
  gt = 6.*exp(4*t)-2.*exp(-t)
end function gt
```

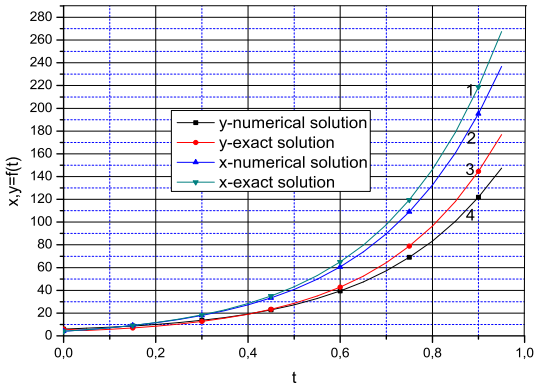

Рис. 8.10:

На рисунке 8.10 приведены графики функций $x = \varphi(t)$ $y = \psi(t)$, полученные по программе 8.7 для уравнения (8.61). Аналитическое решение (8.62) представлено на рисунке кривыми 1 и 3 для $y = \psi(t)$ и $x = \varphi(t)$ соответственно. Численное решение системы обыкновенных дифференциальных уравнений изображено на рисунке кривыми 2 и 4 для y и x соответственно.

Метод Эйлера не обладает достаточной точностью. Поэтому, для численного решения систем ОДУ иожно использовать методы более высокого порядка –методы Гюна, Рунге – Кутта, Адамса, Милна – Симпсона, Хемминга и других.

Пример 8.10 Написать программу решения систем обыкновенных дифференциальных уравнений методом Рунге – Кутта. Воспользуемся уравнениями (8.30) и (8.31). Для решения системы дифференциальных уравнений необходимо применить (8.30) для каждого уравнения:

$$\begin{array}{ll} k_1 = hf(t_k, x_k, y_k), & l_1 = hg(t_k, x_k, y_k), \\ k_2 = hf(t_k + \frac{h}{2}, x_k + \frac{h}{2}k_1, y_k + \frac{h}{2}l_1), & l_2 = hg(t_k + \frac{h}{2}, x_k + \frac{h}{2}k_1, y_k + \frac{h}{2}l_1), \\ k_3 = hf(t_k + \frac{h}{2}, x_k + \frac{h}{2}k_2, y_k + \frac{h}{2}l_2), & l_3 = hg(t_k + \frac{h}{2}, x_k + \frac{h}{2}k_2, y_k + \frac{h}{2}l_2), \\ k_4 = hf(t_k + h, x_k + hk_3, y_k + hl_3), & l_4 = hg(t_k + h, x_k + hk_3, y_k + hl_3). \end{array} \quad (8.64)$$

Окончательные значений искомых функций на новом шаге вычисляются по формулам:

$$\begin{array}{l} x_{k+1} = x_k + (k_1 + 2k_2 + 2k_3 + k_4)/6, \\ y_{k+1} = y_k + (l_1 + 2l_2 + 2l_3 + l_4)/6. \end{array} \quad (8.65)$$

Отметим, что вычисление по формулам (8.64) нужно проводить в указанном порядка – сначала k_1, потом l_1 и так далее. Ниже приведена программа для решения систем ОДУ методом Рунге-Кутта.

Программа 8.8 Программа решения систем ОДУ методом Рунге-Кутта

```
      module two
     integer,parameter::nx = 50
     real(4)::x(nx),y(nx), z(5,nx)
     real(4)::a,b,c,d
     real(4)::h=0.025
     end
    program R_K_System
      use two
      real num(3,nx)
      x(1) = -2.7
      y(1) = 2.8
      t = 0.0
      z(1,1) = 0.
      z(2,1) = x(1)
      z(3,1) = x(1)
      z(4,1) = y(1)
      z(5,1) = y(1)
       do i = 2,nx
         t = t + h
         call R_K(t,x(i-1),y(i-1),x(i),y(i))
         z(1,i) = t
         z(2,i) = x(i)
         z(3,i) = ft(t,x(i),y(i))
         z(4,i) = y(i)
         z(5,i) = gt(t,x(i),y(i))
       enddo

       open(7,file='RK_System.dat')
       write(7,91) (z(1,i),z(2,i),z(3,i),z(4,i),z(5,i),i=1,nx)
91    format(5(1x,f12.7))
       end R_K_System
   subroutine R_K(t,xrk,yrk,xrk1,yrk1)
     use two
     real(4)::k0,k1,k2,k3,l0,l1,l2,l3
     real(4)::t,xrk,yrk
      k0 = h*f(t,xrk,yrk)
      l0 = h*g(t,xrk,yrk)
```

```
    k1 = h*f((t+0.5*h),(xrk+0.5*k0),(yrk+0.5*l0))
    l1 = h*g((t+0.5*h),(xrk+0.5*k0),(yrk+0.5*l0))
    k2 = h*f((t+0.5*h),(xrk+0.5*k1),(yrk+0.5*l1))
    l2 = h*g((t+0.5*h),(xrk+0.5*k1),(yrk+0.5*l1))
    k3 = h*f((t+h),(xrk+k2),(yrk+l2))
    l3 = h*g((t+h),(xrk+k2),(yrk+l2))
    xrk1 = xrk+(k0+2.*k1+2.*k2+k3)/6.
    yrk1 = yrk+(l0+2.*l1+2.*l2+l3)/6.
  end
  real function f(t,x,y)
    real x,y,t
    f = 2.0*x+3.*y
  end function f
  real function g(t,x,y)
    real x,y,t
    g = 2.*x+y
  end
  real function ft(t,x,y)
   real x,y,t
   ft = -69.0/25.0*exp(-t)+3.0/50.0*exp(4.*t)
 end function ft
 real function gt(t,x,y)
   real x,y,t
   gt = 69.0/25.0*exp(-t)+1.0/25.0*exp(4.0*t)
 end function gt
```

На рисунке 8.11 приведены графики изменения x и y в зависимости от t. Точность метода Рунге – Кутта (8.33) существенно выше метода Эйлера, поэтому на графике эти кривые сливаются. На графике кривые 1 и 2 соответствуют численному и аналитическому решениям для x, а 3 и 4 для y. Для того, чтобы показать разницу между точным и аналитическим решениями для x и y на рисунке 8.12 приведены ошибки, то есть $err1 = (x - xt)$ – кривая 1 и $err2 = (y - yt)$ – кривая 2.

8.8. Упражнения

1. Методом Рунге – Кутта найти решение системы уравнений Лотка – Вольтерра (8.57) и (8.58) со следующими значениями коэффициентов: A=2, B=0.02, C=0.002, D=0.8 на интервале [0,5] с шагом 0.1. Начальные данные:

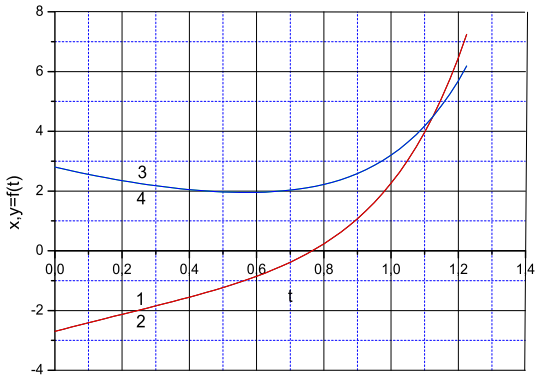

Рис. 8.11:

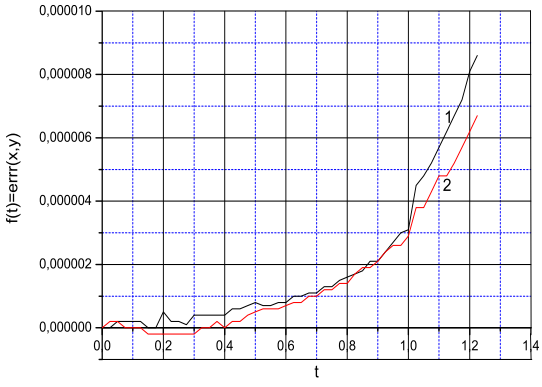

Рис. 8.12:

1. x(0)=3000, y(0)=120;

2. x(0)=5000, y(0)=100.

2. Напишите программу решения системы обыкновенных дифференциальных уравнений методом Адамса.

3. Напишите программу решения системы обыкновенных дифференциальных уравнений с помощью неявного метода.

4. Напишите программу решения системы обыкновенных дифференциальных уравнений с помощью метода Хемминга.

5. С помощью программы решения систем ОДУ методом Рунге – Кутта найти решение системы уравнений

$$\begin{aligned} x' &= x - 4y, \\ y' &= x + y, \end{aligned}$$

с начальными данными $x(0) = 2$, $y(0) = 3$ на отрезке [0,2] с шагом h=0.1. Аналитическое решение:

$$\begin{aligned} x(t) &= -2e^t + 4e^t \cos^2(t) - 12e^t \cos(t)\sin(t), \\ y(t) &= -3e^t + 6e^t \cos^2(t) + 2e^t \cos(t)\sin(t). \end{aligned}$$

6. С помощью программы решения систем ОДУ методом Хемминга найти решение системы уравнений

$$\begin{aligned} x' &= 3x - y, \\ y' &= 4x - y, \end{aligned}$$

с начальными данными $x(0) = 0,2$, $y(0) = 0,5$ на отрезке [0,2] с шагом h=0.2. Аналитическое решение:

$$\begin{aligned} x(t) &= 1/5e^t - 1/10te^t \\ y(t) &= 1/2e^t - 1/5te^t. \end{aligned}$$

7. Найти решение системы уравнений:

$$\begin{aligned} x' &= -3x - 2y - 2xy, \\ y' &= -y + xy, \end{aligned}$$

с начальными данными $x(0) = 4$, $y(0) = 1$ на отрезке [0,8] с шагом h=0.05.

Глава 9

Численные методы решения дифференциальных уравнений в частных производных

9.1. Введение

В последние годы широкое развитие получили методы математического моделирования. Эти методы активно применяются при проектировании и создании ракетно–космических систем, современных летательных аппаратов, при создании новых лекарств, и аппаратов в медицине, расшифровке генома человека, моделировании астрофизических процессов, и целом ряде других задач. Все эти задачи, которые возникают перед учеными, описываются дифференциальными уравнениями в частных производных. Это уравнения Навье – Стокса, Эйлера, Максвелла, Шредингера и так далее. Во всех этих уравнениях существуют функциональные зависимости между пространственными, временными, гравитационными, электромагнитными и другими независимыми переменными и производными от них. Дифференциальные уравнениях в частных производных (ДУЧП) отличаются от обыкновенных дифференциальных уравнений важным обстоятельством, а именно тем, что ДУЧП зависят от нескольких независимых переменных и производным от них, а не от одного независимого переменного. Это обстоятельство позволяет численно моделировать гораздо более сложные процессы, в которые входит несколько независимых переменных и производные от них. Такое большое развитие методы математического моделирования получили в силу ряда обстоятельств. Этому способствовало, во первых, развитие средств вычис-

лительной техники, создание современных многоядерных процессоров и мощных вычислительных кластеров, а во вторых, развитие численных методов для решения возникающих задач.

9.2. Классификация дифференциальных уравнений в частных производных

Определение 9.1 *Если в уравнении искомая функция зависит от* ***нескольких независимых переменных и производных от них****, то такое уравнение называется дифференциальным уравнением в частных производных.*

Частные производные обозначаются так: $\frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^3 u}{\partial x \partial y \partial z}$. Существует эквивалентное обозначение частных производных: u_x, u_{yy}, u_{xyz} соответственно.

В общем виде дифференциальное уравнение в частных производных может быть записано в виде:

$$F(x_1, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n,}, \frac{\partial^2 u}{\partial x_1^2}, \frac{\partial^2 u}{\partial x_1 \partial x_2}, \frac{\partial^n u}{\partial x_1^{p_1}, \partial x_2^{p_2}, \ldots, \partial x_n^{p_n}}) = \\ = f(x_1, \ldots, x_n),$$

Здесь $P_1 + P_2 + \ldots + P_k = n$

Определение 9.2 *Порядком дифференциального уравнения в частных производных называется порядок старшей производной в уравнении (9.1).*

Определение 9.3 *Если $f(x_1, \ldots, x_n) \equiv 0$ – такое уравнение называется однородным, если $f(x_1, \ldots, x_n) \neq 0$ – такое уравнение называется неоднородным.*

Определение 9.4 *Если в (9.1) функция* ***u*** *входит линейно, тогда такое уравнение называется линейным. Если* ***u*** *входит нелинейно, тогда такое уравнение называется нелинейным.*

Определение 9.5 *Уравнение (9.1) называется квазилинейным, если оно линейно относительно старших производных.*

Как и для обыкновенных дифференциальных уравнений, для уравнений в частных производных существует понятия аналитического решения.

Определение 9.6 *Аналитическим решением дифференциального уравнения в частных производных называется функция $u(x_1, x_2, \ldots, x_n)$, которая имеет непрерывные частные производные до n порядка и которая обращает уравнение (9.1) в тождество.*

Общее (аналитическое) решение обыкновенного дифференциального уравнения (8.3) зависит от n независимых постоянных $C_1, C_2, \ldots, C_n$. Общее решение дифференциального уравнения в частных производных зависит от (n-1) произвольной функции. Например, дифференциальное уравнение (9.1), которое называется уравнением переноса

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \tag{9.1}$$

имеет обшее решение, которое зависит от произвольной функции

$$u(t, x) = \varphi(x \pm ct).$$

Линейные дифференциальные уравнения в частных производных второго порядка с двумя независимыми переменными имеют каноническую форму. Общая форма записи такого уравнения имеет вид:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu + G(x, y) = 0 \tag{9.2}$$

В уравнении (9.2) $u = u(x, y), A = A(x, y), B = B(x, y)$ и т.д.

Определение 9.7 *Если в (9.2)* $G(x, y) \equiv 0, \forall x, y$ *– то такое уравнение называется однородным. Если* $G(x, y) \neq 0$ *– такое уравнение называется неоднородным*

Определение 9.8 *Если все коэффициенты уравнения* $A, B, \ldots, G$ *(9.2) постоянны, то такое уравнение называется уравнением с постоянными коэффициентами.*

Все линейные дифференциальные уравнения с частными производными второго порядка относятся к одному из следующих типов:

Определение 9.9 *Если* $B^2 - 4AC > 0$ *– такое уравнение называется* **гиперболическим**.

Определение 9.10 *Если* $B^2 - 4AC = 0$ *– такое уравнение называется* **параболическим**.

Определение 9.11 *Если* $B^2 - 4AC < 0$ *– такое уравнение называется* **эллиптическим**.

Пример 9.1.

1. $$\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 0 \quad \textbf{гиперболическое уравнение}$$

   Здесь $A = 1,\ B = 0,\ C = -1\ \Rightarrow B^2 - 4AC = 0^2 - (4 \cdot 1 \cdot (-1)) = 4 > 0$

2.
$$\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \textbf{параболическое уравнение}$$

Здесь $A = 0,\ B = 0,\ C = 1\ \Rightarrow B^2 - 4AC = 0^2 - 4 \cdot 0 \cdot 1 = 0$

3.
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \textbf{эллиптическое уравнение}$$

Здесь $A = 1,\ B = 0,\ C = 1\ \Rightarrow B^2 - 4AC = 0^2 - 4 \cdot 1 \cdot 1 = -4 < 0$

Уравнения, приведенные в примерах (1. – 3.), называются: 1. – волновым уравнением, 2. – уравнением диффузии (или теплопроводности), 3. – уравнением Лапласа.

Существует много способов решения дифференциальных уравнений в частных производных. Некоторые из них позволяют получить аналитические решения. Другие – только приближенные решения. Однако, только для очень небольшого количества уравнений в частных производных можно получить решение в аналитической форме.

Дадим краткую характеристику основных методов решения УРЧП.

1. **Методы разделения переменных**. В этих методах уравнения с частными производными с n независимыми переменными сводятся к n обыкновенным дифференциальным уравнениям.

2. **Методы интегральных преобразований.** УРЧП с n независимыми переменными сводятся к системе с n-1 независимыми переменными.

3. **Методы преобразования координат.** Исходное УРЧП заменой системы координат сводится к более простым уравнениям.

4. **Методы преобразования независимых переменных.** Вводятся другие независимые переменные, для которых уравнения решаются проще.

5. **Численные методы.** Сведение УРЧП к системе алгебраических уравнений.

6. **Методы теории возмущений.** Исходная нелинейная задача сводится к совокупности линейных задач.

7. **Метод функций Грина.** Вводится система источников и задача решается как суперпозиция этих элементарных источников.

8. **Метод интегральных уравнений.** Дифференциальное уравнение в частных производных сводится к интегральным уравнениям.

9. **Вариационные методы.** Вместо УРЧП решается задача минимизации. Часто оказывается, что функция, доставляющая минимум некоторому функционалу является одновременно решением исходного УРЧП.

10. **Метод разложения по собственным функциям.** Решение УРЧП ищется в виде ряда по собственным функциям.

Несмотря на большой выбор методов решения уравнений с частными производными, только численные методы позволяют получить решение более 90 % задач.

Существует два подхода к описанию дифференциальных уравнений в частных производных: собственно теория дифференциальных уравнений в частных производных и теория уравнений математической физики. Первый из них реализует качественный, аналитический подход. Другой подход харатеризуется более прикладным характером. Уравнениями математической физики называется математический аппарат, который занимается, в основном, задачами математического моделирования и методами их численного решения.

В теории уравнений математической физики приняты некоторые дополнительные соглашения. В них принято определять размерность задачи по количеству физических координат. Если задача зависит от одной пространственной переменной, то такая задача называется одномерной, если от двух: x и y – то такая задача называется двумерной и т.д. Кроме координат задача может зависеть и от других независимых переменных. Так, если задача зависит от времени, то она называется нестационарной, если не зависит – стационарной.

9.2.1. Понятие корректно поставленной задачи

Задача УРЧП является математически корректно поставленной (по Адамару) если:

- Решение задачи существует.

- Решение задачи является единственным.
- Решение непрерывно зависит от начальных и граничных условий.

Корректно поставленной задачей для численного решения будем называть такую задачу, для которой выполняются аналогичные условия:

- Численное решение задачи существует.
- Численное решение задачи является единственным.
- Численное решение непрерывно зависит от начальных и граничных условий.

Пример Адамара некорректно поставленной задачи. В этом примере показано, что решение не всегда непрерывно зависит от начальных условий.

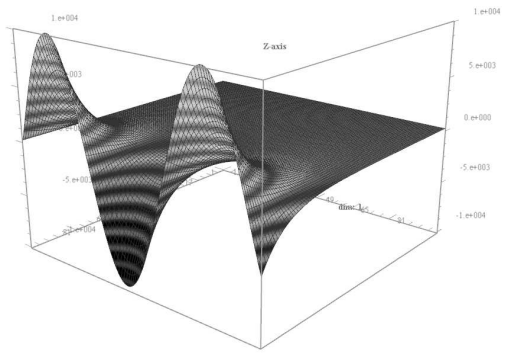

Рис. 9.1:

Пример 9.2 Решить уравнение Лапласа:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \quad x,\ y \in \Omega, \tag{9.3}$$

с граничными условиями:

$$\begin{aligned} u(x,0) &= 0, \\ \frac{\partial u(x,0)}{\partial y} &= \frac{1}{n} sin(nx), n > 0. \end{aligned}$$

Методом разделения переменных [10] можно найти аналитическое решение:

$$u = (\frac{1}{n^2}) \sin(nx) \sinh(ny).$$

На рисунке 9.1 приведено аналитическое решение уравнения (9.3) для n=1.

Если задача корректно поставлена, решение должно непрерывно зависеть от граничных условий. Второе из этих условий имеет вид:

$$\frac{\partial u}{\partial y}(x,0) = \frac{1}{n} \sin(nx)$$

Отсюда следует, что при больших n величина u_y малая. Решение ведет себя при больших n иначе. При $n \to \infty \quad u \to \frac{e^{ny}}{n^2}$. Экспонента растет гораздо быстрее n^2, это означает, что оно неограниченно растет даже при малых y. Однако, из первого граничного условия $u(x,0) = 0$. Это означает, в данном случае отстутствует непрерывность по начальным данным. Таким образом можно сделать вывод, что задача некорректно поставлена.

9.2.2. Начальные и граничные условия для уравнений в частных производных

Как и в случае обыкновенных дифференциальных уравнений частное решение уравнения (9.1) можно найти, задав начальные и граничные условия. Начальные условия задают распределение искомой функции $u(x,y)$ в момент времени $t = t_0$

$$u(x,t_0) = u_0. \tag{9.4}$$

Граничные условия для дифференциальных уравнений в частных производных задаются в следующем виде:

1. **Граничное условие Дирихле.** На границе Ω задается значение искомой функции:

$$u(t,\Omega) = f(t). \tag{9.5}$$

2. **Граничное условие фон Неймана.** На границе Ω задаются производные искомой функции:

$$\frac{\partial u(t,\Omega)}{\partial n} = \varphi(t). \tag{9.6}$$

3. **Граничное условие Робина.** На границе Ω задается комбинация исходной функции и ее производной:

$$\frac{\partial u(t,\Omega)}{\partial n} + u(t,\Omega) = \psi(t). \tag{9.7}$$

Для численного решения уравнения (9.1) с начальными и граничными условиями (9.4) – (9.7) необходимо выполнить следующие шаги:

- Провести дискретизацию исходного дифференциального уравнения в частных производных
- Найти алгоритм решения полученной системы дифференциальных уравнений и решить ее.

Исходные дифференциальные уравнения в частных производных описывают непрерывные процессы в непрерывном пространстве независимых переменных. Нам необходимо корректно преобразовать эти непрерывные функции в систему алгебраических уравнений на дискретной сетке.

Определение 9.12 *Переход от непрерывных уравнений с частными производными к системе алгебраических уравнений на дискретной сетке называется* **дискретизацией.**

9.3. Метод сеток

Будем искать решение уравнения (9.1) с начальными и граничными условиями (9.4) – (9.7) на прямоугольнике:

$$R = \{(x,t) : \ 0 \le x \le a, \ 0 \le t \le b\}.$$

Разобъем прямоугольник R на сетку, состоящую из $(N-1)(M-1)$ прямоугольников. Стороны прямоугольников (в общем случае не одинаковые) обозначим: $\triangle x_i, \ \triangle y_n, \ 0 \le i \le N, \ 0 \le n \le M$. Таким образом, вместо непрерывной функции $u(x,t)$ мы ввели сеточную функцию $u(i\triangle x, n\triangle y)$. Положения точек узлов сетки внутри области будет определяться (для равномерной сетки) значениями индексов i, n. Разностные уравнения будем записывать для произвольного внутреннего узла $\{i,n\}$ введенной сетки. На границах расчетной области Ω, то есть при $\{i = 1, \ i = N\}$ будем задавать граничные условия (9.5) – (9.7).

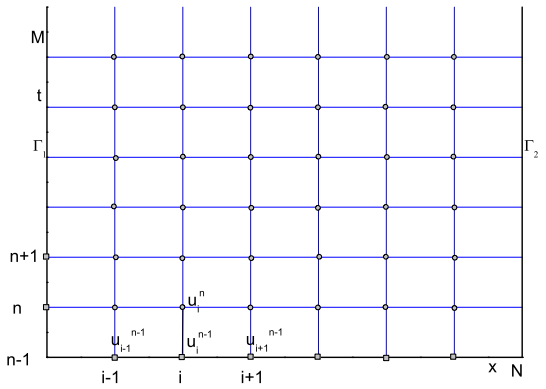

Рис. 9.2:

На рисунке 9.2 изображена прямоугольная сетка, которая покрывает расчетную область R. u - сеточная функция, определенная в узлах $0 \leqslant i \leqslant M$ и $0 \leqslant n \leqslant N$. Будем обозначать узлы сетки следующим образом: u_i^n. Γ_1 и Γ_2 - левая и правая границы расчетной области. На нижнем слое задаются начальные условия.

Определение 9.13 *Совокупность узлов, используемых в сеточном уравнении называется* **шаблоном**.

На рисунке 9.2 изображен шаблон типа "перевернутое Т", который образуют узлы разностной сетки $u_{i-1}^{n-1},\ u_i^{n-1},\ u_{i+1}^{n-1},\ u_i^n$.

Определение 9.14 *Функция дискретного аргумента, определенная на разностной сетке, называется* ***сеточной функцией***.

9.3.1. Дискретизация производных

Дискретизация производных по времени. Рассмотрим одно из простых уравнений в частных производных – уравнение переноса.

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0. \tag{9.8}$$

Представим производную по времени в виде конечной разности:

$$\frac{\partial u}{\partial t} \approx \frac{\left(u_i^{n+1} - u_i^n\right)}{\triangle t}. \tag{9.9}$$

В этом уравнении используется информация из двух слоев: неизвестного $n+1$ и известного n.

Производную по пространству представим в виде:

$$\frac{\partial u}{\partial x} \approx \frac{(u_i^n - u_{i-1}^n)}{\triangle x}. \tag{9.10}$$

Подставив (9.9) и (9.10) в (9.8), получим:

$$u_i^{n+1} = u_i^n - \frac{c\triangle t}{\triangle x}(u_i^n - u_{i-1}^n). \tag{9.11}$$

В данном случае мы использовали для дискретизации уравнения (9.8) шаблон "левый уголок". Можно использовать схему "правый уголок":

$$u_i^{n+1} = u_i^n - \frac{c\triangle t}{\triangle x}(u_{i+1}^n - u_i^n), \tag{9.12}$$

или "центральную разность":

$$u_i^{n+1} = u_i^n - \frac{c\triangle t}{2\triangle x}(u_{i+1}^n - 2u_i^n + u_{i-1}^n). \tag{9.13}$$

9.3.2. Точность процесса дискретизации

Процесс дискретизации осуществляется с некоторой точностью. То есть, при проведении этого процесса, мы получаем некоторую ошибку. Формулы односторонних разностей (9.11), (9.12) являются точными для линейных функций. Формула центральной разности (9.13) является точной для квадратичных функций. Во всех остальных случаях существует ошибка конечно – разностного представления.

Эту ошибку можно оценить, разложив искомую функцию в ряд Тейлора в окрестности того узла, где оценивается величина ошибки. Разложим схему "левый уголок"в ряд Тейлора:

$$\begin{aligned}
&u_i^{n+1} \approx u_i^n + \triangle t(\tfrac{\partial u}{\partial t})_i^n + \tfrac{1}{2}\triangle t^2(\tfrac{\partial^2 u}{\partial t^2})_n^{n+\theta_1} + \dots, \\
&u_{i-1}^n \approx u_i^n - \triangle x(\tfrac{\partial u}{\partial x})_i^n - \tfrac{1}{2}\triangle x^2(\tfrac{\partial^2 u}{\partial x^2})_{i-\theta_2}^n + \dots, \\
&\tfrac{1}{\triangle t}(u_i^{n+1} - u_i^n) = \tfrac{c}{\triangle x}(u_i^n) = (\tfrac{\partial u}{\partial t} + c\tfrac{\partial u}{\partial x})_i^n + O(\triangle t, \triangle x).
\end{aligned}$$

Таким образом, схемы "левый"и "правый" уголок являются схемами первого порядка точности по x и t. Можно показать, что схема "центральная разность", имеет порядок точности $O(\triangle t, \triangle x^2)$.

Существует два основных способа повышения точности расчетов. Один из них состоит в уменьшении шага $\triangle x$. Другой состоит в применении конечно – разностных схем более высокого порядка точности.

Эти соображения приводят к понятию экономической эффективности конечно – разностных схем. Если использовать первый способ, то для достижения требуемой точности, необходимо будет сделать во много раз больше арифметических операций, чем во втором случае.

Еще один важный вопрос состоит в том, в какой степени численное решение дифференциального уравнения в частных производных будет близко к точному решению. Сходимость численного решения к точному очень трудно установить прямым путем. Поэтому приходится использовать обходной путь. Потребуем, чтобы выполнялось требование согласованности алгебраических уравнений, полученных в процессе дискретизации с исходным дифференциальным уравнением в частных производных.

Определение 9.15 *Если при разложении в ряд Тейлора процесс дискретизации может быть обращен и получена исходная система дифференциальных уравнений, то полученная система алгебраических уравнений называется* **согласованной**.

Кроме того, алгоритм получения решения алгебраических уравнений должен быть устойчивым.

Определение 9.16 *Если при решении системы алгебраических уравнений возникающие неконтролируемые возмущения затухают, то такое решение называется* **устойчивым**.

Определение 9.17 *Решение алгебраических уравнений, аппроксимирующих заданное дифференциальное уравнение в частных производных, называется* ***сходящимся****, если это решение стремится к точному решению УРЧП, для любого значения независимой переменной при стремлении размера ячеек сетки к 0:* $u_i^n \to \overline{u}(x,t)$, *при* $\triangle x, \triangle t \to 0$, *здесь* $\overline{u}(x,t)$ *– точное решение.*

Теперь можно сформулировать теорему Лакса об эквивалентности.

Теорема Лакса. *Пусть имеется корректно поставленная задача с начальными условиями и конечно – разностная аппроксимация этой задачи. Тогда устойчивость является необходимым и достаточным условием сходимости численного решения к точному.*

Эта теорема очень важна, так как она дает возможность доказать сходимость численного решения к точному при наличии согласованности и устойчивости.

9.4. Уравнение переноса

Основные идеи и методы построения разностных схем будем изучать на модельном уравнении переноса (9.8) в несколько обобщенном виде.

$$\frac{\partial u}{\partial t} + \frac{\partial F(u)}{\partial x} = 0 \tag{9.14}$$

Эти методы лежат в основе решения современных сложных задач гидроаэродинамики, газовой динамики, механики деформируемого твердого тела, задач физики, химии, биологии и т.д. Для уравнения (9.14) зададим начальные условия:

$$\begin{aligned} &u(0,x) = u_0(x), \\ &u(t,0) = u_1,\ (x,t) \in \Omega, \\ &u(t,1) = u_2,\ 0 \le x \le 1,\ 0 \le t \le T. \end{aligned} \tag{9.15}$$

Для данной уравнения с начальными условиями приведем несколько численных методов решения, напишем программы и сравним численной решение с аналитическим. Данная задача позволяет сохранить в себе все особенности решения прикладных задач гиперболического типа, а с другой стороны, является достаточно простой для понимания и реализации. На примере этой модельной задачи можно наглядно показать основные методы, разностные схемы и технологию использования разностных схем для уравнений математической физики. При решении задач гидроаэродинамики могут встречаться особенности: например, разрывные решения, связанные с ударными волнами и контактными разрывами. Будем считать, что задача (9.14) – (9.16) корректно поставлена и удовлетворяет условию Липшица.

Для численного решения задач математической физики используются два вида разностных схем: явные и неявные.

Определение 9.18 *Если дифференциальный оператор по пространственным производным, аппроксимируется на нижних временным слоях, то такая разностная схема называется* **явной.**

Определение 9.19 *Если для аппроксимации дифференциального оператора используются значения на верхнем временном слое, то такия схема называются* **неявной.**

Введем прямоугольную сетку в области Ω. По оси абсцисс будем откладывать x, а по оси ординат t (рисунок 9.2).

Таким образом, мы сформулировали корректно поставленную задачу Коши для уравнения гиперболического типа (9.14). Несмотря на то,

что классификация дифференциальных уравнений с частными производными определена только для уравнений второго порядка – принято полагать, что уравнение (9.14) тоже относится к гиперболическому типу.

Будем полагать, что $\varphi''(u) \geq 0$. Рассмотрим, следуя [11] – [12], линейные однородные разностные схемы для уравнения (9.14). Будем изучать только двух и трехслойные разностные схемы. В общем виде их можно записать в виде:

$$u_i^{n+1} = \sum_{l \in S_1} \alpha_l u_l^n + \sigma \sum_{n \in S_2} u_j^{n-1} \tag{9.16}$$

Уравнение (9.16) имеет частное решение:

$$u_m^{n+1} = e^{i(\omega(n+1)\tau - kmh)} = q^{n+1} e^{ikmh}, \quad q = e^{i\omega t}, \tau = \triangle t, h = \triangle x. \tag{9.17}$$

Характеристическое уравнение имеет вид:

$$q^2 - q \sum_{l \in S_1} \alpha_i e^{ikh(m-l)} - \sigma \sum_{j \in S_2} \beta_j e^{ikh(m-j)} = 0. \tag{9.18}$$

Здесь $S - \{S_1, S_2\}$ шаблон разностной схемы.

Если корни разностного уравнения (9.18) $|q_{1,2}| \equiv 1$, то такая схема недиссипативна. Это означает, что сама разностная схема не порождает источников (стоков). Если $|q_{1,2}| \leqslant 1$, – схема диссипативна, если $|q_{1,2}| > 1$, – схема неустойчива.

Самые простые разностные схемы решения уравнения (9.14) мы рассмотрели в предыдущем параграфе. Это схемы "левый" и "правый" уголок и "центральная разность".

Определение 9.20 *Условие устойчивости Куранта – Фридрихса – Леви (КФЛ). Для уравнения (9.14) существует условие устойчивости КФЛ:* $\triangle t \leqslant K \frac{\triangle x}{c}$. *K – коэффициент, который называется числом Куранта.*

Если c=const, то схема (9.18):

1. Устойчива при числах Куранта $0 < K \leqslant 1$.
2. Дает точное решение при $K = 1$ на равномерной сетке.

9.4.1. Схема Лакса

Рассмотрим схему **Лакса** для решения уравнения (9.14).

$$\frac{u_i^{n+1} - \overline{u}}{\triangle t} + \frac{F(u_{i+1}^n) - F(u_{i-1}^n)}{x_{i+1} - x_{i-1}} = 0, \tag{9.19}$$

где $\overline{u} = \frac{u^n_{i+1}+u^n_{i-1}}{2}$.

Схема Лакса обладает свойством монотонности, условно аппроксимирует решение и имеет первый порядок аппроксимации на рвномерной сетке. Определим понятие **монотонности** разностной схемы. Борис и Бук [20] дают следующее определение монотонности:

Определение 9.21 *Разностная схема, которая не увеличивает число экстремумов по сравнению с точным решением, называется* ***монотонной.***

Приведем еще одно опреление монотонности разностной схемы:

Определение 9.22 *Разностная схема называется* ***монотонной*** *если из условия* $u^n_{i+1} + u^n_i \geqslant 0$ *следует* $u^{n+1}_{i+1} + u^{n+1}_i \geqslant 0$ *для всех* i.

Шаблон разностной схемы:

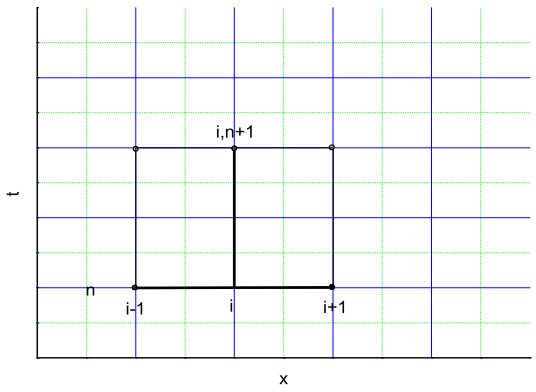

Рис. 9.3:

Если c=const, то схема устойчива при $0 < K \leqslant 1$ и дает точное решение при $k = 0$ на равномерной сетке.

9.4.2. Схема Лакса – Вендроффа.

Схема **Лакса – Вендроффа** – это схема предиктор – корректор:

- **Предиктор**:

$$\frac{u^{n+1/2}_{i-1/2} - \overline{u}}{\triangle t} + \frac{F(u^n_i) - F(u^n_{i-1})}{x_i - x_{i-1}} = 0, \quad \overline{u} = \frac{u^n_i - i^n_{i-1}}{2}.$$

- **Корректор:**

$$\frac{u^{n+1} - u_i^n}{\triangle t} + \frac{F(u_{i+1/2}^{n+1/2}) - F(u_{i-1/2}^{n+1/2})}{x_{i+1/2} - x_{i-1/2}} = 0.$$

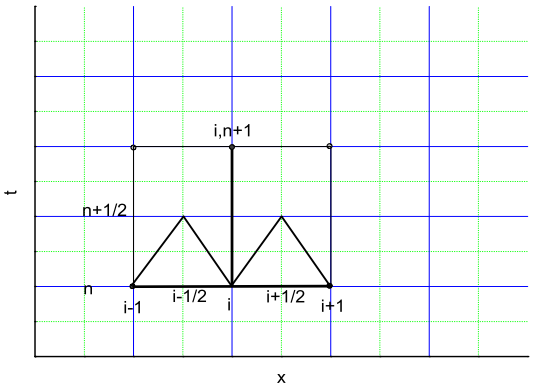

Рис. 9.4:

Схема Лакса-Вендроффа обладает следующими свойствами:

- имеет второй порядок аппроксимации на регулярной сетке,
- не обладает свойствами монотонности.

Если $F(u) = cu, \quad c = const$, то схема:

- устойчива при $0 < K \leqslant 1$.
- дает точное решение при $kK1$.

Характеристическое уравнение для схемы Лакса-Вендроффа:

$$q = 1 + r^2 \cos kh - ri \sin kh - r^2.$$

9.4.3. Схема "крест"

В этой двухшаговой схеме первый шаг делается по схеме Лакса:

$$\frac{u_i^1 - \overline{u}}{\triangle t} + \frac{F(u_{i+1}^0) - F(u_{i-1}^0)}{2\triangle x} = 0$$

здесь $\overline{u} = \frac{u^0_{i+1}+u^0_{i-1}}{2}$.

Дальнейшие вычисления проводятся по формуле:

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\triangle t} + \frac{F(x_{i+1}^n) - F(u_{i-1}^n)}{2\triangle x} = 0.$$

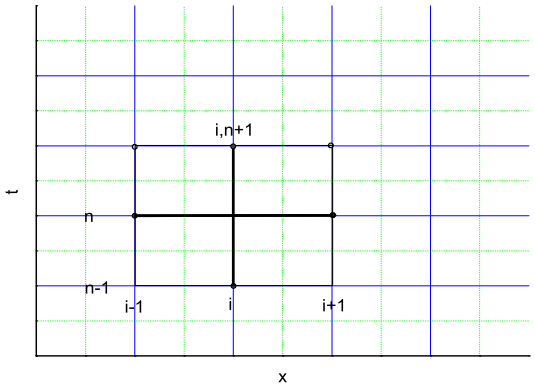

Рис. 9.5:

Схема "крест":

- имеет второй порядок аппроксимации на регулярной сетке;
- не обладает свойством монотонности;

- дает точное решение при числе Куранта K $\leqslant 1$. Если $F(u) = cu, \quad c = const \rightarrow$ схема "крест" устойчива при $0 < Kleqslant1$. Характеристическое уравнение для схемы "крест"

$$q^2 - 2kri \sin kh - 1 = 0.$$

Корни этого уравнения

$$q_{1,2} = -ri \sin kh \pm \sqrt{1 - r^2 \sin^2 kh}$$

если $|r| \leqslant 1 \Rightarrow |q_{1,2}| = \sqrt{1 - r^2 \sin^2 kh + r^2 \sin^2 kh} \equiv 1 \Rightarrow$ схема недиссипативна.

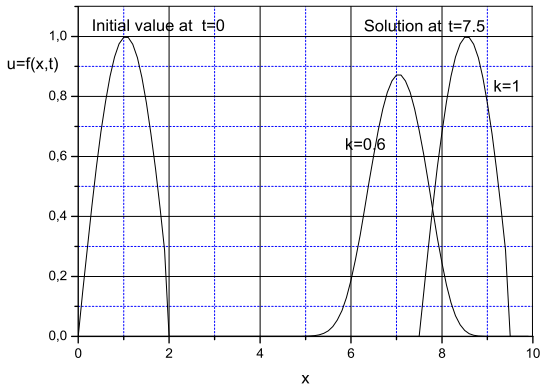

Рис. 9.6:

На рисунке 9.6 приведено решение, полученное по схеме Лакса - Вендроффа, с числом Куранта K=1.0 и K=0.6. На рисунке приведено начальное значение u (слева) и вычисленное к моменту времени t=7.5. Начальное значение задавалось на 1/5 общей длины, по формуле $u = sin(1.5x)$. Отметим влияние диссипации на поведение решения. Общая площадь всех трех графиков одинакова, но влияние диссипации приводит к "расплыванию" решения.

Ниже приведена программа, которая реализует методы "левый уголок", Лакса, Лакса - Вендроффа и "крест". В программе необходимо изменить параметр num для выбора нужной схемы – num=1,2,3,4 - schemes upwind, Lax, Lax-Vendroff, crest. Число Куранта задается с помощью переменной K. В программе K=1.

Программа 9.1 Программа расчета уравнения переноса методами "левый уголок", Лакса, Лакса -Вендроффа, "крест"

```
    program advection
      integer,parameter::nx=100,nt=75
      real,dimension(nx)::x,u1,u2,u3
      real::dx=0.1,c=1.0,dt,K=1.
!     num=1,2,3,4 - schemes upwind, Lax, Lax-Vendroff, crest
      num = 3
      dt = K*dx/c

     do m = 1,nx
        x(m) = (m-1)*dx
```

```
      enddo
        u1(1:nx/5) = sin(1.5*x(1:nx/5))
        u2 = u1

      do m = 1,nt
       select case (num)
         case (1)
          do i = 2,nx
            u2(i) = u1(i)-c*dt*(u1(i)-u1(i-1))/dx
          enddo
          u1 = u2
         case (2)

          do i = 2,nx-1
            u2(i) = 0.5*(u1(i+1)+u1(i-1))-0.5*c*dt*(u1(i+1)- &
                    u1(i-1))/dx
          enddo
          u1 = u2
         case(3)
!  Predictor
          do i = 2,nx-1
            u2(i) = 0.5*(u1(i)+u1(i-1)) - 0.5*c*dt*(u1(i)- &
                    u1(i-1))/dx
          enddo
!  Corrector
          do i = 2,nx-1
            u3(i) = u1(i)-c*dt*(u2(i+1)-u2(i))/dx
          enddo
           u1 = u2
           u2 = u3
         case(4)

          do i =2,nx-1
            u2(i) = 0.5*(u1(i+1)+u1(i-1))-0.5*c*dt*(u1(i+1)- &
                    u1(i-1))/dx
          enddo

          do i =2,nx-1
            u3(i) = u1(i)-c*dt*(u2(i+1)-u2(i-1))/dx
```

```
          enddo
           u1 = u2
           u2 = u3
         end select

      enddo
       u1(1:nx/5) = sin(1.5*x(1:nx/5))
       open(7,file='scheme.dat')
       write(7,91) (x(i),u1(i),i=1,nx)
 91    format(2(2x,f11.6))
    end program advection
```

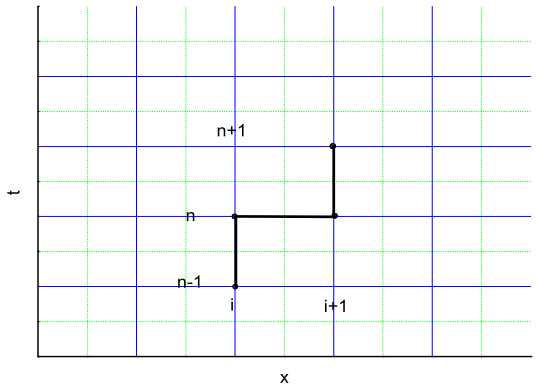

Рис. 9.7:

9.4.4. Схема "кабаре".

Шаблон схемы "кабаре"приведен на рисунке 9.7. Первый шаг делается по схеме "уголок":

$$\frac{u_i^1 - u_i^0}{\triangle t} + \frac{F(u_i^0) - F(u_{i-1}^0)}{\triangle x} = 0.$$

Дальнейшие вычисления осуществляются по формуле:

$$\frac{1}{2}\Big((1+\varepsilon)\frac{u_i^{n+1} - u_i^n}{\triangle t} + (1-\varepsilon)\frac{u_{i-1}^{n+1} - u_{i-1}^n}{\triangle t}\Big) + \frac{F(u_i^n) - F(u_{i-1}^n)}{\triangle x} = 0.$$

Здесь ε зависит от числа Куранта:

$$\varepsilon = K(1-r)(1-2r)^2.$$

При $\varepsilon = 1$ схема "кабаре"переходит в схему "левый уголок".

Упражнение. Напишите программу, реализующую схему "кабаре".

9.5. TVD и ENO схемы

Рассмотрим некоторые современные разностные схемы для решения задач с большими градиентами. Это разностные схемы, реализующие методы коррекции потоков Бориса - Бука [20], схемы TVD и ENO. Основные идеи этих методов будем изучать на основе уравнения переноса (9.14). Объединим схемы (9.11) и (9.12) в одну:

$$u_i^{n+1} = u_i^n - K \begin{cases} u_{i+1}^n - u_i^n, & c < 0, \\ u_i^n - u_{i-1}^n, & c \geqslant 0. \end{cases} \tag{9.20}$$

Здесь K – число Куранта (Определение 9.20). Если ввести обозначения $a^+ = \frac{1}{2}(a + |a|)$ и $a^- = \frac{1}{2}(a - |a|)$ уравнение (9.20) можно представить в виде:

$$u_i^{n+1} = u_i^n - \frac{\triangle t}{\triangle x}(a^-(u_{i+1}^n - u_i^n+) + a^+(u_i^n - u_{i-1}^n)). \tag{9.21}$$

Введем потоковую форму представления схем "левый" уголок и "правый" уголок. Для этого введем две функции $f_{i+1/2}^{n+1/2}$ и $f_{i-1/2}^{n-1/2}$.

$$u_i^{n+1} = u_i^n - K(f_{i+1/2}^n - f_{i-1/2}^n). \tag{9.22}$$

Из уравнений (9.21) и (9.22) можно получить потоковые формы представления уравнения переноса:

$$\begin{aligned} f_{i+1/2}^n &= \frac{1}{2}((u_{i+1}^n + u_i^n) - c(u_{i+1}^n - u_i^n)), \\ f_{i-1/2}^n &= \frac{1}{2}((u_i^n + u_{i-1}^n) - c(u_i^n - u_{i-1}^n)). \end{aligned} \tag{9.23}$$

В общем виде уравнение (9.23) можно представить в виде [21]:

$$u_i^{n+1} = u_i^n - K(\overline{f}_{i+1/2} - \overline{f}_{i-1/2}),$$

здесь $\overline{f}_{i\pm1/2} = (1-\omega)f_{i\pm1/2} + \omega f_{i\pm1/2},\ 0 \leqslant \omega \leqslant 1.$

Приведенные выше разностные схемы имели первый порядок точности и сохраняли свойство монотонности (смотри **Определение 9.22**). Для создания разностных схем, которые одинаково хорошо работают в областях малых и больших градиентов необходимо создать более сложные, гибридные разностные схемы. Такие схемы имеют разный порядок точности в областях с большими и малыми градиентами решения. Рассмотрим такую схему. Предположим, что скорость с положительна с>0.

$$u_i^{n+1} = u_i^n - K(u_i^n - u_{i-1}^n) + \frac{\gamma}{2}K(1-K)(u_{i+1}^n - 2u_i^n + u_{i-1}^n) = 0. \quad (9.24)$$

В приведенном выше уравнении K - число Куранта, а γ - параметр гибридности [22]:

$$\gamma = \begin{cases} 1, & |u_{i+1}^n - 2u_i^n + u_{i-1}^n| < \lambda |u_i^n - u_{i-1}^n|, \\ 0, & |u_{i+1}^n - 2u_i^n + u_{i-1}^n| \geqslant \lambda |u_i^n - u_{i-1}^n| \end{cases} \quad (9.25)$$

В уравнении (9.25), при $\lambda = 0$ схема имеет первый порядок точности, а при $\lambda > 0$ второй порядок точности. Параметр γ называется ограничителем (лимитером) разностной схемы (9.24).

Аналогично (9.24) можно построить разностные схемы более высокого порядка. Например, уравнение (9.26) имеет третий порядок аппроксимации:

$$u_i^{n+1} = u_i^n - K(u_i^n - u_{i-1}^n) + \frac{\gamma}{2}K(1-K)\{(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \quad (9.26)$$
$$+(u_{i+1}^n - 3u_i^n + 3u_{i-1}^n - u_{i-2}^n)\}.$$

9.5.1. Метод Бориса – Бука (коррекции потоков)

В работе [20], для улучшения параметров разностных схем предложен метод коррекции потоков. Метод коррекции потоков основан на двух щаговой схеме предиктор – корректор. Н первом этапе в схему вводится дополнительная вязкость, которая увеличивает устойчивость схемы. На втором этапе – корректор, в областях гладкого решения из схемы убирается введенная диффузия.

Шаг предиктор:

$$\widetilde{u}_i = u_i^n - \frac{1}{2}\left[\varepsilon_{i+1/2}(u_{i+1}^n + u_i^n) - \varepsilon_{i-1/2}(u_i^n - u_{i-1}^n)\right] + \quad (9.27)$$
$$+\left[\nu_{i+1/2}(u_{i+1}^n + u_i^n) - \nu_{i-1/2}(u_i^n - u_{i-1}^n)\right] =$$
$$= u_i^n - \frac{\triangle t}{\triangle x}\left[f_{i-1/2} - f_{i+1/2}\right].$$

Здесь $\varepsilon_{i+1/2}$ и $\nu_{i+1/2}$ являются безразмерными коэффициентами численной диффузии, которая вводится на этапе предиктор.

Шаг корректор можно записать в виде:

$$u_i^{n+1} = \widetilde{u}_i + (\widetilde{f}_{i-1/2} - \widetilde{f}_{i+1/2}).$$

Значения $\widetilde{f}_{i-1/2}$ и $\widetilde{f}_{i+1/2}$ являются антидиффузионными потоками.

$$\begin{aligned} \widetilde{f}_{i+1/2} &= \mu(\widetilde{u}_{i+1} - \widetilde{u}_i), \\ \widetilde{f}_{i-1/2} &= \mu(\widetilde{u}_i - \widetilde{u}_{i-1}) \end{aligned} \tag{9.28}$$

Здесь μ - коэффициент антидиффузии.

Шаг корректор можно записать в обобщенном виде:

$$u_i^{n+1} = \widetilde{u}_i + \mu(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \widetilde{\mu}(\widetilde{u}_{i+1}^n - 2\widetilde{u}_i^n + \widetilde{u}_{i-1}^n) \tag{9.29}$$

Где μ и $\widetilde{\mu}$ - коэффициенты антидиффузии.

9.5.2. Схема TVD (Total Variation Diminition)

Схемы TVD (схемы с уменьшением полной вариации) позволяют более точно аппроксимировать дифференциальные уравнения в частных производных для задач с большими градиентами параметров и разрывными решениями. Рассмотрим схему с уменьшением полной вариации на примере схемы Лакса - Вендроффа [23]. Как известно (см.параграф 9.4.2), схема Лакса - Вендроффа немонотонная схема второго порядка на регулярной сетке и состоит из двух шагов - предиктор и корректор.

$$u_i^{n+1} = u_i^n - K(u_i^n - u_{i-1}^n) - (f_{i+1/2}^n - f_{i-1/2}^n). \tag{9.30}$$

$$\begin{aligned} f_{i+1/2} &= K(1 - c\triangle t)(\overline{u}_{i+1}^{n+1} - \overline{u}_i^n), \\ f_{i-1/2} &= K(1 - c\triangle t)(\overline{u}_i^{n+1} - \overline{u}_{i-1}^n), \end{aligned}$$

здесь K - число Куранта. Эта схема превратится в монотонную схему (левый уголок) если в уравнении (9.30) опустить член $(f_{i+1/2}^n - f_{i-1/2}^n)$. Это говорит о том, что в оригинальной схеме $\varphi(r_i)$ антидиффузионные потоки велики, что и приводит к появлению осцилляций. Если ввести ограничители этих антидиффузионных потоков, то полученная разностная

схема будет обладать свойством ограничения полной вариации. Ограничители можно ввести разными способами, например:

$$\widetilde{f}_{i+1/2} = \varphi(r_i)K(1 - c\triangle t)(\overline{u}_{i+1} - \overline{u}_i). \tag{9.31}$$

В уравнении (9.31) функция $\varphi(r_i)$ ограничивает поток. Значение r_i является показателем "гладкости" решения и вычисляется по формуле:

$$r_i = \frac{u_i - u_{i-1}}{u_{i+1} - u_i}. \tag{9.32}$$

В области гладкого решения $r_i \approx 1$; в области больших градиентов $r_i \approx 0$. Выбор функции $\varphi(r_i)$ осуществляется таким образом, чтобы на каждом шаге по времени полная вариация была ограниченной. Выбор такого ограничителя $\varphi(r_i)$ позволяет отнести данную модификацию метода Лакса - Вендроффа к классу схем TVD.

Запишем условие полной вариации:

$$TV(u^n) = \sum_{i,n=0}^{\infty} |u_{i+1}^n - u_i^n|.$$

Тогда схемой с ограничением полной вариации можно назвать такую схему, для которой выполняется неравенство $TV(u^{n+1}) \leqslant TV(u^n)$. Отметим, что ограничение полной вариации является более слабым условием, чем условие монотонности.

Выбор ограничителя обусловлен необходимостью корректного ограничения верхней границы TVD диапазона [24]. Ограничитель должен минимизировать потоки вблизи разрывов и не искажать решение в области гладкого решения. Рассмотрим некоторые ограничители для TVD схем.

$$\begin{gathered} 0 < \varphi(r_i) \leqslant \min(2r_i, 2), \text{ для } r_i > 0, \\ \varphi(r_i) = 0, \text{ для } r_i \leqslant 0. \end{gathered} \tag{9.33}$$

Для того, чтобы схема обеспечивала второй порядок аппроксимации необходимо, что бы в уравнении (9.33) функция $\varphi(1) = 1$.

Более точный ограничитель можно задать следующим образом:

$$\varphi(r_i) = \begin{cases} \min(2, r_i), & r > 1, \\ \min(2r_i, 1), & 0 < r \leqslant 1, \\ 0, & r \leqslant 0. \end{cases} \tag{9.34}$$

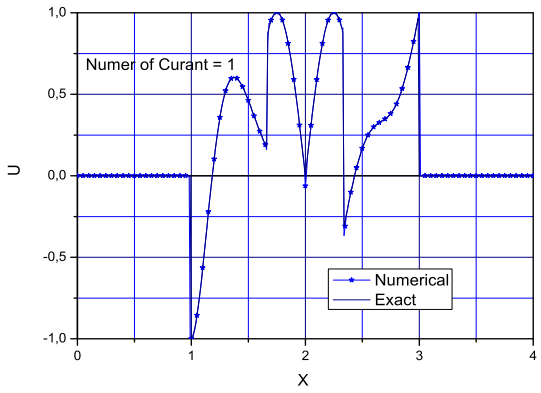

Рис. 9.8:

Пример 9.3 Рассмотрим решение уравнения переноса методом TVD с ограничителями (9.34). Начальные условия зададим на отрезке {-1.0 - 4.0}:

$$u_0 = \begin{cases} -x\sin(\frac{3}{2}\pi x^2), & -1 \leqslant x < -\frac{1}{3}, \\ |\sin(2\pi x)|, & |x| \leqslant \frac{1}{3}, \\ 2x - 1 - \frac{1}{6}\sin(3\pi x), & \frac{1}{3} < x < 1. \end{cases} \tag{9.35}$$

Такие начальные условия были предложены Хартеном [25] для проверки разностных схем. На рисунке Рис. 9.8 приведен график функции (9.35) через 200 шагов по времени. Число Куранта KR=1. Можно отметить хорошее совпадение численного решения, полученного по методу TVD (на рисунке обозначено звездочками), с точным решением (сплошная линия).

Программа 9.2 Программа решения уравнения переноса методом TVD

```
module one
  integer,parameter::nx=401, nt=200,nw=4
  real(4)::u0(nx), u1(nx), u2(nx), u3(nx), x(nx), f(nx), &
        v(nw,nx), fi(nx)
  real(4)::dx=0.01, dt, c=1.0, Pi= 3.141529, KR=1.0, r, &
        epsr = 1.0E-07
end module one
```

```
    program TVD
      use one

      dt = KR*dx/c
!  initial data
      do i = 1,nx
        x(i) = -1.0 + (i-1)*dx
      enddo
      do i = 1,nx     !nx/5
         xx = x(i)

        if( xx <= -1.0/3.0 ) then
           u1(i) = -xx*sin(3.0/2.0*Pi*xx*xx)
        elseif( xx <= 1.0/3.0 ) then
           u1(i) = abs(sin(2.0*Pi*xx))
        elseif( xx <= 1.0 ) then
           u1(i) = 2.0*xx-1.0-1.0/6.0*sin(3.0*Pi*xx)
        endif
      enddo
      u0 = u1
      u2 = u1
      v(1,:) = x
      v(2,:) = u2
! TVD Схема Лакса - Вендроффа с ограничителями
! Шаг по времени
     do m = 1,nt
! Шаг предиктор
      do i = 2,nx-1
        u2(i) = 0.5*(u1(i)+u1(i-1))-0.5*KR*(u1(i)-u1(i-1))
      enddo

! Шаг корррректор ! Вычисление потоков
      do i = 2,nx-1
        f(i) = 0.5*KR*(1.0-c*dt)*(u2(i+1)-u2(i))
      enddo
! Ограничители потоков
      do i = 2,nx-1
        r = (u2(i)-u2(i-1)+epsr)/(u2(i+1)-u2(i)+epsr)
       if( r>1.0 ) then
```

```
        fi(i) = min(2.0,r)
       elseif( r <= 1.0 .and. r>0.0 ) then
        fi(i) = min(2.0*r,1.0)
       else
        fi(i) = 0.0
       endif
        f(i) = fi(i)*KR*(1-c*dt)*(u2(i+1)-u2(i))
      enddo

      do i = 2,nx-1
        u3(i) = u1(i)-KR*(u2(i+1)-u2(i))-(f(i)-f(i-1))
      enddo
       u1 = u2
       u2 = u3
     enddo

       xx = x(1)
       v(3,:) = u3

     do k = 1,nt
       ll = k + c*dt*nt/dx+12
      if( ll >= nx ) exit
        v(4,ll) = u0(k)
     enddo

      open(7,file='TVD1.txt')
     do l = 1,nx
        write(7,91) (v(m,l),m=1,nw)
     enddo
      close(7)

      write(*,*) dt,nt
 91   format(4(2x,e15.6))
    end program TVD
```

На Рис. 9.9 приведено численное решение уравнения переноса с начальными условиями (9.35), полученное через 200 шагов по времени. Число Куранта KR=0.9 Совпадение численного решения, полученного по методу TVD (на рисунке обозначено звездочками), с точным решением (сплошная линия) несколько хуже. В приведенной выше программе

9.2 уравнение (9.32) несколько изменено.

$$r_i = \frac{u_i - u_{i-1} + \varepsilon}{u_{i+1} - u_i + \varepsilon}. \tag{9.36}$$

Значение $\varepsilon = 10^{-6} \div 10^{-11}$ выбирается для того, чтобы избежать нежелательного "шума" .

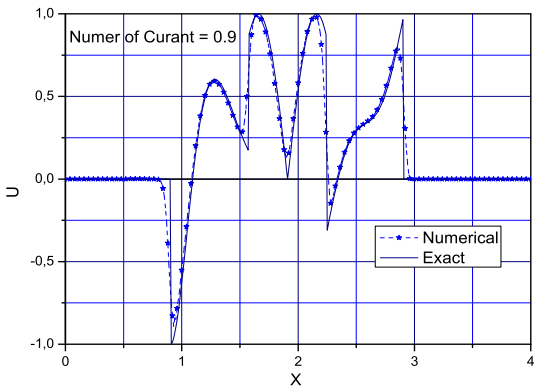

Рис. 9.9:

Еще одну разностную схему, которая построена по схеме TVD, можно записать в виде:

$$u_i^{n+1} = u_i^n - K\{(u_i^n - u_{i-1}^n) + [\xi_{i+1/2}(u_{i-1}^n - u_i^n) - \xi_{i-1/2}(u_i^n - u_{i-1}^n)]\}, \tag{9.37}$$

Из (9.37), с учетом условия $\xi_{i\pm1/2} \geqslant 0$ можно получить:

$$u_i^{n+1} = u_i^n + \frac{u_i^n - u_{i-1}^n}{\triangle x} \cdot \left(1 + \frac{\xi_{i+1/2}}{2}\frac{u_{i+1}^n - u_i^n}{u_i^n - u_{i-1}^n} - \frac{\xi_{i-1/2}}{2}\right) = 0.$$

Если выражение в круглых скобках будет больше нуля, тогда можно утверждать, что схема будет монотонной. Порядок схемы можно задать, изменяя коэффициент $\xi_{i\pm1/2}$:

$$\xi(r_i) = \begin{cases} 0 & r_i \leqslant 0, \\ \frac{[d+1(1-\delta)]}{(d+b)(1-\delta)} r_i, & 0 < r_i < 1 - \Delta, \\ \frac{d+br_i}{d+b}, & |r_i - 1| \leqslant \delta, \\ \frac{(d+b(1-\delta))-2d\delta}{(d+b)(1-\delta)} r_i, & 1 + \Delta < r_i \leqslant \delta, \\ \leqslant 2, & r_i \geqslant 2. \end{cases} \tag{9.38}$$

В уравнении (9.38) b, d, δ константы, а $0 < \Delta < 1$. Если выбрать $b + d \neq 0$ мы получим второй порядок аппроксимации схемы. Если выбрать b=2/3, а d=1/3 - мы получим третий порядок аппроксимации всех точках, кроме точек разрыва функций.

9.5.3. Схема ENO (Essentialy Non Oscillatory)

Схемы ENO (особенно не осциллирующие) используются, как и схемы TVD, для построения разностных схем для задач с большими градиентами величин. Такие схемы были впервые предложены Хартеном [25]. Схемы ENO используются для того, чтобы во первых повысить порядок аппроксимации до второго и выше, а во вторых сдлать схему монотонной без дополнительного введения исскуственной или схемной вязкости.

Рассмотрим разностную схему для уравнения переноса $u_t + cu_x = 0$ в потоковой форме:

$$u_i^{n+1} = u_i^n - K(\overline{f}_{i+1/2} - \overline{f}_{i-1/2}). \tag{9.39}$$

Потоки в уравнении (9.39) будем вычислять по формулам:

$$\overline{f}_{i+1/2} = \begin{cases} \alpha u_{i-1}^n + (1-\alpha-\beta)u_i^n + \beta_{i+1}^n, & a \geqslant 0, \\ \alpha u_{i+2}^n + (1-\alpha-\beta)u_{i+1}^n + \beta u_i^n, & a < 0. \end{cases} \tag{9.40}$$

$$\overline{f}_{i-1/2} = \begin{cases} \alpha u_{i-2}^n + (1-\alpha-\beta)u_{i-1}^n + \beta_i^n, & a \geqslant 0 \\ \alpha u_{i+1}^n + (1-\alpha-\beta)u_i^n + \beta u_{i-1}^n, & a < 0, \end{cases} \tag{9.41}$$

В уравнениях (9.40) и (9.41) - α и β коэффициенты, которые необходимо определить. Если разложить разностную схему (9.39) в ряд Тейлора в окрестности узла $\{u_i^n\}$ получим:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = \left[\frac{\triangle x}{2}|c|(1 + 2\alpha - 2\beta) - \frac{c^2 \triangle t}{2}\right]\frac{\partial^2 u}{\partial x^2}. \tag{9.42}$$

Из (9.42) следует, что при $\alpha = \beta = 0$ схема (9.39) будет иметь первый порядок аппроксимации, при $\alpha = 0,\ \beta = 1/2$ - второй порядок аппроксимации.

Схемная вязкость в уравнениях (9.39) - (9.41) будет равна нулю при выполнении равенства:

$$\beta = \frac{1}{2}\left(1 - \frac{|c|\triangle x}{\triangle t}\right).$$

Для того, чтобы обеспечить условие монотонности разностной схемы типа ENO необходимо выполнение условия:

$$\frac{1-K}{2(2-K)}\triangle_{i-1/2}u \leqslant \triangle_{i+1/2}u \leqslant \frac{2(1+K)}{K}\triangle_{i-1/2}u. \qquad (9.43)$$

Здесь K число Куранта,

$$\triangle_{i+1/2}u = u_{i+1} - u_i,$$
$$\triangle^2_{i+1/2}u = \triangle_{i+1}u - \triangle_i u.$$

В зависимости от условия "гладкости" решения осуществляется переключением потоков: если $(c \cdot \triangle u \cdot \triangle^2 u)_{i+1/2} \geqslant 0$

$$\overline{f}_{i+1/2} = \begin{cases} \frac{3-K}{2}u^n_i - \frac{1-K}{2}u^n_{i-1}, & c \geqslant 0 \\ \frac{3-K}{2}u^n_{i+1} - \frac{1-K}{2}u^n_{i+2}, & c < 0. \end{cases}$$

Если $(c \cdot \triangle u \cdot \triangle^2 u)_{i+1/2} < 0$, тогда

$$\overline{f}_{i+1/2} = \frac{1-cK}{2}u^n_{i+1} + \frac{1+cK}{2}u^n_i.$$

9.6. Гиперболические уравнения.

Гиперболические уравнения описывают волновые и колебательные движения сплошной среды. Для задач математической физики гиперболические уравнения имеют, как правило, выделенную координату - время. Если искомая функция зависит от времени – такое уравнение называется нестационарным, если не зависит – стационарным. Наиболее распространенным гиперболическим уравнением является волновое уравнение.

9.6.1. Волновое уравнение

Волновое уравнение (9.44) описывает различные волновые процессы. В частности, такими уравнениями описываются колебания закрепленной с двух сторон струны, распространение. В одномерном случае, то есть при наличии только одной пространственной переменной, зависящей от времени, волновое уравнение можно записать в виде:

$$\frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0, \quad 0 < x < a, \quad 0 < t < b. \qquad (9.44)$$

Зададим начальные условия для уравнения (9.44):

$$u(x,0) = \varphi(x), \quad 0 \leqslant x \leqslant a,$$
$$\frac{\partial u(x,0)}{\partial t} = \psi(x), \quad 0 < x < a. \tag{9.45}$$

Граничные условия для уравнения (9.44) зададим в виде:

$$u(0,t) = 0,$$
$$u(a,t) = 0, \quad 0 \leqslant t \leqslant b. \tag{9.46}$$

Волновое уравнение описывает волновые движения среды – в частности колебания закрепленной с двух сторон струны. Аналитическое решение задачи (9.44) можно получить, используя разложение в ряд Фурье.

Рассмотрим численное решение данной задачи. Для этого введем прямоугольную сетку $\Re = \{ (x,t) : 0 \leqslant x \leqslant a,\ 0 \leqslant t \leqslant b\}$, которая состоит из прямоугольников с размерами по осям x, t : $\triangle x$, $\triangle t$ соответственно. Дискретизацию волнового уравнения проведем, используя формулы центральной разности:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{u(x,t+\triangle t) - 2u(x,t) + u(x,t-\triangle t)}{\triangle t^2} + O(\triangle t^2). \tag{9.47}$$

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{u(x+\triangle x,t) - 2u(x,t) + u(x-\triangle x,t)}{\triangle x^2} + O(\triangle x^2). \tag{9.48}$$

Введем обозначения $x_{i+1} = x_i + \triangle x$ и $t^{n+1} = t^n + \triangle t$.

Используя введенные обозначения и пренебрегая величинами $O(\triangle t^2)$ и $O(\triangle x^2)$ получим:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\triangle t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\triangle x^2} \tag{9.49}$$

Обозначим $\lambda = c\triangle t/\triangle x$, тогда уравнение (9.49) можно записать в виде:

$$u_i^{n+1} = 2(1-\lambda^2)u_i^n + \lambda^2(u_{i+1}^n + u_{i-1}^n) - u_i^{n-1}. \tag{9.50}$$

Уравнение (9.50) позволяет перейти от известных значений на n временном слое к неизвестным величинам на новом n+1 временном слое. Такие численные схемы называются **явными**, так как решение на новом временном слое находится из алгебраического уравнения (9.50) за один шаг. Но при этом приходится вводить ограничения на шаг по времени: $\triangle t \leqslant \triangle x/c$, которое называется условием устойчивости Куранта – Фридрихса – Леви. Без этого условия разностная схема для решения волнового уравнения становится неустойчивой.

9.6.2. Начальные и граничные условия

Для того, чтобы можно было начинать вычисления по уравнению (9.50) необходимо задать начальные и граничные условия. Из уравнения (9.50) следует, что нам необходимо иметь начальные значения, заданные на двух слоях по времени: n и $n-1$. Начальные условия обычно задаются на одном слое. Используем разложение в ряд Тейлора функции $u(x,t)$ в окрестности граничной точки $(x,0)$ для вычисления следующего слоя по времени:

$$u(x_i, \triangle t) = u(x_i, 0) + \frac{\partial u(x_i, 0)}{\partial t}\triangle t + O(\triangle t^2). \tag{9.51}$$

Применим начальные условия (9.45). Так как $u(x_i, 0) = \varphi(x_i) = \varphi_i$ и $u_t(x_i, 0) = \psi(x_i) = \psi_i$ отсюда можно получить необходимые значения на втором временном слое.

$$u_i^{(2)} = \varphi_i + \triangle t\psi_i \quad \text{для} \quad i = 2, 3, \ldots, n-1. \tag{9.52}$$

При использовании формулы (9.52), возникающая ошибка не уменьшается с течением времени и, кроме того имеет тенденцию распространяться по всей расчетной области. Для уменьшения ошибки рекомендуется выбирать достаточно малым шаг $\triangle t$.

Если функция $\varphi(x)$ имеет вторую производную $\varphi''(x)$, то $u_{xx}(x,0) = \varphi''(x)$. В этом случае можно использовать разложение в ряд Тейлора:

$$u_{tt}(x_i, 0) = c^2 u_{xx}(x_i, 0) = c^2\varphi''(x_i) = c^2\frac{\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}}{\triangle x^2} + O(\triangle x^2). \tag{9.53}$$

Разлагаем в ряд Тейлора в окрестности точки $x = x_i$ и оставляем члены до второго порядка включительно:

$$u(x, \triangle t) = u(x, 0) + u_t(x, 0)\triangle t + \frac{u_{tt}(x, 0)\triangle t^2}{2!} + O(\triangle t^3). \tag{9.54}$$

Подставляя (9.52) и (9.53) в (9.54) получим:

$$u(x_i, \triangle t) = \varphi_i + \triangle t\varphi_i + \frac{c^2\triangle t^2}{2\triangle x^2}(\varphi_{i+1} - 2\varphi_i + \varphi_{i-1}) + O(\triangle x^2, \triangle t^3). \tag{9.55}$$

Можно упростить уравнение (9.55), используя обозначение $r = c\triangle t/\triangle x$. Отсюда, получим окончательно выражение для вычисления начального

значения на втором слое.

$$u_i^{(2)} = (1 - r^2)\varphi_i + \triangle t\psi_i + \frac{r^2}{2}(\varphi_{i+1} + \varphi_{i-1}), \quad \text{для} \quad i = 2, 3, \ldots, n-1. \tag{9.56}$$

Пример 9.4 Решить волновое уравнение

$$\frac{\partial^2 u(x,t)}{\partial t^2} = 4\frac{\partial^2 u(x,t)}{\partial x^2} \tag{9.57}$$

$0 \leqslant x \leqslant 3$, $0 \leqslant t \leqslant 1.5$.

Граничные условия:

$$u(0,t) = 0, \quad u(3,t) = 0, \quad 0 \leqslant t \leqslant 1.5. \tag{9.58}$$

Начальные условия:

$$u(x,0) = \varphi(x) = \sin(\pi x) + \sin(2\pi x), \quad 0 \leqslant x \leqslant 3,$$
$$u_t(x,0) = \psi(x) = 0, \quad 0 \leqslant x \leqslant 3. \tag{9.59}$$

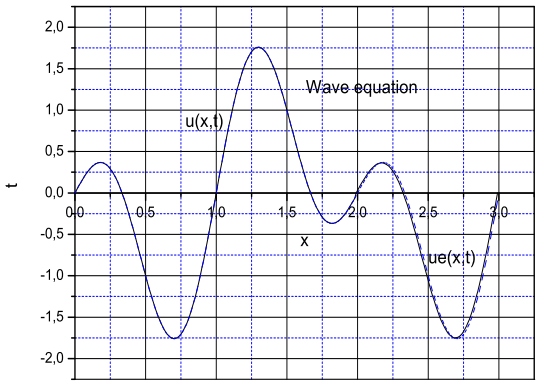

Рис. 9.10:

Аналитическое решение уравнения (9.57) записывается в виде [17]:

$$u(x,t) = \sin(\pi x)\cos(2\pi t) + \sin(2\pi x)\cos(4\pi t). \tag{9.60}$$

Будем решать уравнение (9.57) с помощью (9.56). На рисунке 9.10 приведено численное решение волнового уравнения (9.57) с начальными (9.58)

и граничными (9.59) условиями. На графике численное решение представлено сплошной линией. Пунктирной линий показано аналитическое решение (9.60). На отрезке [0,2] оно практически совпадает с численным решением, а на отрезке [2,3] расхождение с численным решением становится более заметным.

Программа 9.3 Программа для решения волнового уравнения

```
    program Lex5
!   Решение волнового уравнения
      integer,parameter::mx=300,nt=100
      real::u0(mx),u1(mx),u2(mx),ut(mx),x(mx),z(2,mx),y(mx,nt)
      real::dx=0.01,dt=0.005,Pi=3.141592653589793, &
            lambda,c=2.,t=0.

     do i = 1,mx
      x(i) = (i-1)*dx
     enddo

     do i = 1,mx
       u0(i) = f(x(i))
     enddo
     lambda = c*dt/dx

     do i = 2,mx-1
       u1(i) = (1.-lambda**2)*f(x(i))+dt*g(x(i))+ &
                0.5*lambda**2*(f(x(i+1))+f(x(i-1)))
     enddo

     do n = 1,nt
       do i = 2,mx-1
        u2(i) = 2.*(1.-lambda**2)*u1(i)+lambda**2*(u1(i+1)+ &
                u1(i-1))-u0(i)
       enddo
       u0=u1
       u1=u2
       y(:,n) = u2
      enddo
     t = dt*nt
     ut = sin(Pi*x)*cos(2.*Pi*t)+sin(2.*Pi*x)*cos(4.*Pi*t)
```

```
     z(1,:) = u2(1:mx)
     z(2,:) = ut(1:mx)
     open(7,file='Wave.dat')
     write(7,91) (x(i),z(1,i),z(2,i),i=1,mx)
     close(7)
 91  format(3(f11.6))
     write(*,*)
    end program Lex5

    function f(x)
      real:: f,x,Pi=3.14159265358979323846264 3
      f = sin(Pi*x) + sin(2.*Pi*x)
    end

    function g(x)
      real:: g,x,Pi=3.14159265358979323846264 3
      g = 0.
    end
```

9.7. Параболические уравнения

Рассмотрим параболическое уравнение, записанное в одномерном нестационарном случае:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\lambda \frac{\partial u}{\partial x}\right) \tag{9.61}$$

Здесь λ – коэффициент теплопроводности или диффузии. Уравнение (9.61) описывает процессы распространения тепла или процесс диффузии. Диффузия – это процесс проникновения молекул одного вещества в молекулы другого. Допустим, что λ – константа. Проведем дискретизацию уравнения (9.61) по времени и по пространству. Получим:

$$\frac{u_i^{n+1} - u_i^n}{\triangle t} = \lambda \frac{\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)}{\triangle x^2} \tag{9.62}$$

Полученная схема называется схемой ВВЦП (по Времени Вперед, Центральная по Пространству). В англоязычной литературе она называется схемой FTCS (Forward Time, Centered Space). Узлы, которые связаны между собой соотношением (9.62), изображены на рисунке 9.11.

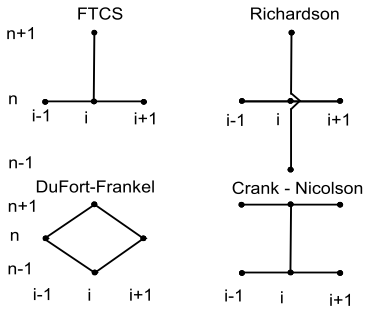

Рис. 9.11:

Кроме шаблона для схемы ВВЦП, на рисунке 9.11 приведены шаблоны для разностных схем Ричардсона, Дюфорта – Франкела и Кранка – Николсона.

Схема ВВЦП согласована с исходным дифференциальным уравнением (9.61). Это можно проверить, разложив уравнение (9.62) в ряд Тейлора в окрестности узла $\{i, n\}$:

$$\Big[\frac{\partial u}{\partial t} - \lambda \frac{\partial^2 u}{\partial x^2}\Big]_i^n + E_i^n = 0. \tag{9.63}$$

Ошибка аппроксимации, таким образом определяется из уравнения

$$E_i^n = \Big[\frac{\triangle t}{2}\frac{\partial^2 \overline{u}}{\partial t^2} - \lambda \frac{\triangle x^2}{12}\frac{\partial^4 \overline{u}}{\partial x^4}\Big]_i^n + O(\triangle t^2, \triangle x^4). \tag{9.64}$$

Схема ВВЦП имеет первый порядок точности по времени и второй по пространству, при $\triangle t, \triangle x \to 0$.

Введем обозначение $s = \frac{\lambda \triangle t}{(\triangle x)^2}$. Используя введенные обозначения уравнение (9.62) можно записать в виде:

$$u_i^{n+1} = s(u_{i+1}^n + u_{i-1}^n) + (1 - 2s)u_i^n \tag{9.65}$$

Анализ устойчивости схемы ВВЦП приводит к уравнению:

$$\xi = 1 - \frac{4\lambda \triangle t}{\triangle x^2} \sin^2\big(\frac{k \triangle x}{2}\big). \tag{9.66}$$

Условие невозрастания возмущения $|\xi| \leqslant 1$ приводит к необходимости выполнения следующего условия устойчивости:

$$\frac{\lambda \triangle t}{(\triangle x)^2} \leqslant 0.5. \tag{9.67}$$

Физическая интерпретация условия устойчивости (9.67) означает, что за время $\triangle t$ диффузионный поток не должен пройти расстояние большее, чем одна ячейка.

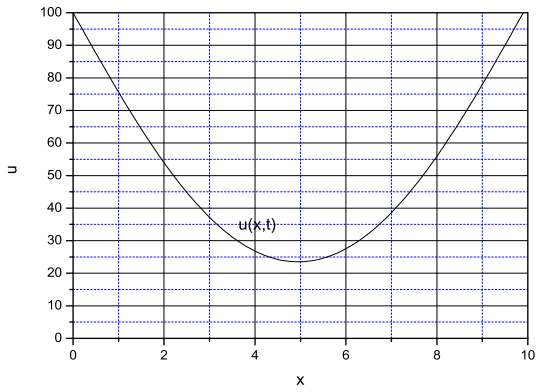

Рис. 9.12:

Пример 9.5 Решить одномерное уравнение теплопроводности

$$\frac{\partial u(x,t)}{\partial t} - \lambda \frac{\partial^2 u(x,t)}{\partial x^2} = 0. \tag{9.68}$$

на отрезке [0,10].

Начальные условия:

$$u(x,0) = 0. \tag{9.69}$$

Граничные условия:

$$\begin{aligned} u(0,t) &= 100, \\ u(10,t) &= 100. \end{aligned} \tag{9.70}$$

На рисунке 9.12 приведено численное решение уравнения (9.68) с начальными (9.69) и граничными(9.70) условиями на момент времени t=0.5.

Ниже приведена программа решения уравнения теплопроводности (диффузии) в одномерном случае методом ВВЦП.

Программа 9.4 Программа для решения уравнения теплопроводности

```
program Thermal
 implicit none
   integer,parameter::nx = 100,mt=1000
   integer::i,j
   real::   x(nx),u(2,nx),dx=0.1,s=0.5,lambda=10.,dt

   dt = s*dx*dx/lambda
   do  i = 1,nx
     x(i) = (i-1)*dx
   enddo

   do j = 1,mt
     u(1,1)  = 100.
     u(1,nx) = 100.
     u(2,1)  = 100.
     u(2,nx) = 100.

    do i = 2,nx-1
      u(2,i) = s*(u(1,i+1)+u(1,i-1)) + (1.-2.*s)*u(1,i)
    enddo
      u(1,:) = u(2,:)
   enddo
   open(7,file='Thermal.dat')
   write(7,91) (x(i),u(1,i),i=1,nx)
91  format(2(2x,f11.6))
  end program Thermal
```

9.7.1. Схемы Ричардсона и Дюфорта – Франкела

Метод ВВЦП, для аппроксимации производной по времени, использует схему первого порядка точности. По пространству эта схема имеет второй порядок точности. Для того, чтобы усовершенствовать схему ВВЦП, и сделать ее второго порядка точности как по времени, так и по простран-

ству, Ричардсон предложил следующую модификацию.

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\triangle t} - \frac{\lambda(u_{i-1}^n - 2u_i^n + u_{i+1}^n)}{2\triangle x^2} = 0, \tag{9.71}$$

На рисунке 9.11 изображен шаблон схемы Ричардсона. Схема (9.71) имеет второй порядок точности как по времени, так и по пространству. Анализ устойчивости схемы (9.71) показывает, что схема Ричардсона безусловно неустойчива при $s > 0$. Если в (9.71) заменить u_i^n на $0.5(u_i^{n+1} - u_i^{n-1})$, то получится устойчивая разностная схема:

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\triangle t} - \frac{\lambda(u_{i-1}^n - (u_i^{n-1} + u_i^{n+1}) + u_{i+1}^n)}{\triangle x^2} = 0. \tag{9.72}$$

Уравнение (9.72) называется разностной схемой Дюфорта – Франкела. На рисунке 9.11 изображен шаблон разностной схемы Дюфорта – Франкела. Данная схема является трехслойной, так как в нее входят искомые функции из трех временных слоев $n-1,\ n,\ n+1$. Оставим в левой части неизвестное значение u_i^{n+1}, остальные члены перенесем в правую часть уравнения. Если ввести обозначение $s = \frac{\lambda\triangle t}{\triangle x^2}$, тогда уравнение можно записать в виде:

$$u_i^{n+1} = \Big(\frac{1-2s}{1+2s}\Big)u_i^{n-1} + \frac{2s}{1+2s}(u_{i-1}^n + u_{i+1}^n). \tag{9.73}$$

Из (9.73) видно, что при $s = 0.5$ схема Дюфорта – Франкела совпадает со схемой ВВЦП. Для проведения расчетов по трехслойным схемам необходимо задавать начальные данные на первых двух временных слоях. В таких случаях, первый временной слой рассчитывается по двухслойной схеме, например ВВЦП, дальнейший расчет осуществляется по (9.73).

Если разложить (9.73) в ряд Тейлора в окрестности узла $\{i, n\}$, получим:

$$\Big[\frac{\partial\overline{u}}{\partial t} - \lambda\frac{\partial^2\overline{u}}{\partial x^2} + \lambda\Big(\frac{\triangle t}{\triangle x}\Big)\frac{\partial^2\overline{u}}{\partial t^2}\Big] + O(\triangle t^2, \triangle x^2) = 0. \tag{9.74}$$

Для обеспечения согласованности (**Определение 9.15**) необходимо, чтобы $\frac{\triangle t}{\triangle x} \to 0$, при $\triangle t \to 0$ и $\triangle x \to 0$, то есть необходимо, чтобы $\triangle t \ll \triangle x$. Анализ ошибок аппроксимации показывает [19], что если $s = (1/12)^{1/2}$, то схема Дюфорта – Франкела дает ошибку аппроксимации порядка $O(\triangle x^4)$.

Упражнение. Напишите программу, реализующую метод Дюфорта – Франкела для задачи (9.68) с начальными и граничными условиями (9.69) и (9.70).

9.7.2. Неявная схема Кранка – Николсона

Условие устойчивости (9.67) для явных разностных схем накладывает существенное ограничение на шаг по времени. Напомним, что явной называется такая разностная схема, в которой можно выразить неизвестные значения на $n+1$ слое через уже найденные значения на n слое. Неявные разностные схемы не имеют таких ограничений на шаг по времени, поэтому рассмотрим неявную разностную схему для решения уравнения диффузии. В неявных схемах используются такой шаблон разностной схемы (смотри рисунок 9.11), в который на новом временном слое входит не одна неизвестная величина, а несколько. Рассмотрим разностную схему, предложенную Кранком и Николсоном. Для лучшей аппроксимации дифференциального оператора $\frac{\partial u}{\partial t}$ используется формула центральной разности:

$$u_t(x, t + \frac{\triangle t}{2}) = \frac{u(x, t + \triangle t) - u(x, t)}{\triangle t} + O(\triangle t^2) \tag{9.75}$$

Аналогично аппроксимируем дифференциальный оператор $u_{xx}(x, t + \triangle t/2)$:

$$\begin{aligned} u_{xx}(x, t + \triangle t/2) = \frac{1}{2\triangle x^2}(u(x - \triangle x, t) \quad + \\ u(x + \triangle x, t + \triangle t) + u(x - \triangle x, t) \quad - \\ 2u(x, t) + u(x + \triangle x, t)) + O(\triangle x^2). \end{aligned} \tag{9.76}$$

Подставив (9.75) в (9.76) в уравнение (9.61) и пренебрегая остаточными членами $O(\triangle x^2)$ и $O(\triangle t^2)$, получим:

$$-su_{i-1}^{n+1} + 2(1+s)u_i^{n+1} - su_{i+1}^{n+1} = 2(1-s)u_i^n + s(u_{i-1}^n + u_{i+1}^n), \tag{9.77}$$

здесь $i = 1, \ldots, N-1$. Слева от знака равенства находятся неизвестные величины, справа все величины известны. Получившаяся система уравнений является ленточной системой с трехдиагональной матрицей. Для ее решения можно воспользоваться методом простой (3.4) итерации или методом Гаусса – Зейделя (3.6)

Если положить s=1, то уравнение (9.77) примет вид:

$$-u_{i-1}^{n+1} + 4u_i^{n+1} - u_{i+1}^{n+1} = u_{i-1}^n + u_{i+1}^n. \tag{9.78}$$

Уравнение (9.78) можно представить в виде ленточной матрицы, с шириной ленты равной трем. Граничные условия используются в самом

первом и в последнем уравнениях:

$$\begin{vmatrix} 4 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & -1 & 4 \end{vmatrix} \cdot \begin{vmatrix} u_2^{n+1} \\ u_3^{n+1} \\ \dots \\ u_{N-1}^{n+1} \end{vmatrix} = \begin{vmatrix} 2\lambda_1 + u_3^n \\ u_2^{n+u_4^n} \\ \dots \\ u_{N-2}^n + 2\lambda_2 \end{vmatrix}$$

Пример 9.6 Методом Кранка – Николсона решить двумерное уравнение уравнение

$$u_t(x,y,t) = u_{xx}(x,y,t), \quad a \leqslant x \leqslant b,\ a \leqslant y \leqslant b,\ 0 \leqslant t \leqslant t^*,$$

с начальным условием:

$$u_x(x,0) = f(x) = 0, \quad t = 0,\ 0 \leqslant x \leqslant 1.$$

и граничными условиями:

$$u(a,y,t) = 100,$$

$$u(b,y,t) = 100,$$

$$u(x,a,t) = 100,$$

$$u(x,b,t) = 100.$$

На рисунке 9.13 приведен график функции u(x,y) к моменту времени t^*.

Ниже приведена программа решения двумерного уравнения диффузии неявным методом Кранка–Николсона.

Программа 9.5 Программа для метода Кранка - Николсона

```
Program Crank_Nicolson
  integer,parameter::nx=100,ny=100
  real::alpha_x(nx),alpha_y(ny),betta_x(nx), &
        betta_y(ny),T_x(nx,ny),T_y(nx,ny),T(nx,ny)
  real::dt=0.1,dx=0.1,dy=0.1,lambda_x=0.01, &
           lambda_y=0.01,s_x,s_y
  real::a_x(nx),b_x(nx),c_x(nx),a_y(ny),b_y(ny),c_y(ny)
   s_x = (lambda_x*dt)/(dx*dy)
   s_y = (lambda_y*dt)/(dx*dy)

    a_x = -s_x
```

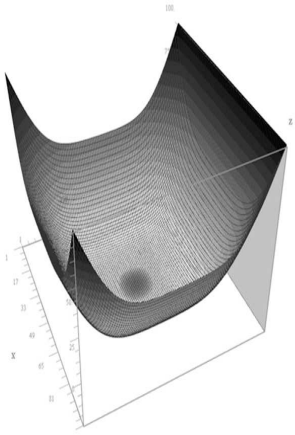

Рис. 9.13:

```
        b_x = 2.*s_x+1.
        c_x = -s_x
        a_y = -s_y
        b_y = 2.*s_y +1.
        c_y = -s_y
!  Граничные условия
        T_x(1,:) = 100.
        T_x(nx,:) = 100.
        T_y(:,1) = 100.
        T_y(:,ny) = 100.
!  Цикл по времени
      do m = 1,1000
!  Вычисление прогоночных коэффициентов по X
      do j = 1,ny
        betta_x(1) = T_x(1,j)
        do i = 2,nx-1
          alpha_x(i) = -c_x(i)/(a_x(i)*alpha_x(i-1)+b_x(i))
          betta_x(i) = (T_x(i,j)-betta_x(i-1)*a_x(i))/ &
                       (a_x(i)*alpha_x(i-1)+b_x(i))
        enddo
       do i = nx,2,-1
        T_x(i-1,j) = alpha_x(i-1)*T_x(i,j)+betta_x(i-1)
       enddo
      enddo
```

```
!  Вычисление прогоночных коэффициентов по  Y
      do i = 1,ny-1
         betta_y(1) = T_y(i,1)
        do j = 2,ny
          alpha_y(j) = -c_y(j)/(a_y(j)*alpha_y(j-1)+b_y(j))
          betta_y(j) = (T_y(i,j)-betta_y(j-1)*a_y(j))/ &
                       (a_y(j)*alpha_y(j-1)+b_y(j))
        enddo
       do j = ny,2,-1
         T_y(i,j-1) = alpha_y(j-1)*T_y(i,j)+betta_y(j-1)
       enddo
      enddo

      do i = 1,nx
        do j = 1,ny
         if( T_x(i,j) == 0. ) then
           T(i,j) = T_y(i,j)
         elseif(T_y(i,j) == 0. ) then
           T(i,j) = T_x(i,j)
         else
           T(i,j) = 0.5*(T_x(i,j) + T_y(i,j))
         endif
        enddo
      enddo
      enddo
    end Program Crank_Nicolson
```

Упражнение Методом Кранка – Николсона решить одномерное уравнение диффузии

$$u_t(x,t) = u_{xx}(x,t), \quad 0 \leqslant x \leqslant 1,\ 0 \leqslant t \leqslant 0.1,$$

с начальным условием:

$$u_x(x,0) = f(x) = \sin(\pi x) + \sin(3\pi x), \quad t = 0,\ 0 \leqslant x \leqslant 1,$$

и граничными условиями:

$$u(0,t) = \varphi_1(t) = 0, \quad x = 0,\ 0 \leqslant x \leqslant 0.1,$$

$$u(1,t) = \varphi_2(t) = 0, \quad x = 1,\ 0 \leqslant x \leqslant 0.1.$$

Решение, полученное методом Кранка – Николсона должно совпадать с аналитическим решением, которое дается выражением (9.79):

$$u(x,t) = \sin(\pi x)e^{-\pi^2 t} + \sin(3\pi x)e^{-9\pi^2 t}. \qquad (9.79)$$

9.7.3. Нелинейные уравнения

Рассмотренные выше задачи были достаточно простыми линейными задачами. При создании математических моделей обычно приходится решать нелинейные задачи. Рассмотри простейшее нелинейное уравнение, которое можно получить из уравнения переноса (9.1), если положить в нем c=u.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0. \qquad (9.80)$$

Уравнение (9.80) называется невязким уравнением Бюргерса. Существует вязкое уравнение Бюргерса:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \mu\frac{\partial^2 u}{\partial x^2}. \qquad (9.81)$$

В уравнении (9.81) μ – динамическая вязкость. Первое и второе слагаемые в левой части этого уравнения являются соответственно нестационарным и конвективным членами, а в правой части стоит вязкий член. Если вязкий член не равен нулю, то уравнение (9.81) параболическое; если же он равен нулю, то в уравнении остаются лишь нестационарный и нелинейный конвективный члены. Уравнения типа (9.80) и (9.81) могут использоваться для описания некоторых аэродинамических задач.

Решение уравнений (9.80) и (9.81) может осуществляется с помощью различных разностных схем. Рассмотрим модификацию метода ВВЦП (9.62). Применяя разностную схему (9.62) для уравнения (9.81) получим:

$$u_i^{n+1} = u_i^n - 0.5\triangle t\frac{u_i^n(u_{i+1}^n - u_{i-1}^n)}{\triangle x} + \mu\triangle t\frac{(u_{i+1}^n - 2u_i^n + u_{i-1}^n)}{(\triangle x)^2} \qquad (9.82)$$

Решение нелинейного уравнения существенно отличается от решения уравнения переноса. Передний фронт волны с течением времени становится круче, а задний более пологим. Это связано с тем, что скорость движения переднего фронта волны нелинейно увеличивается с ростом u.

На рисунке 9.14 приведено начальное положение волны – штрихпунктирная линия и положение волны на момент времени t=0.65. Сплошной

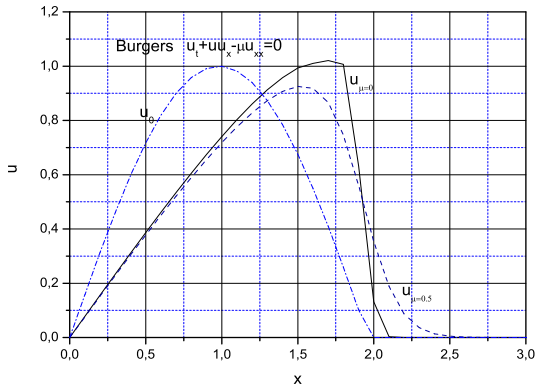

Рис. 9.14:

линией представлено решение без вязкости и пунктирной линией – решение со значением динамической вязкости равным $\mu = 0.05$. Численное решение было получено с помощью программы 9.5.

Программа 9.6 Программа решения уравнения Бюргерса

```
program Burgers
  implicit none
  integer,parameter::nx=100,nt=65
  real,dimension(nx)::x, u1, u2, u3, u4, z(4,nx)
  real:: dt, dx, mu, F
  integer i,j,k
   mu = 0.05
   dx = 0.1
   dt = 0.01
 do i = 1,nx
   x(i) = (i-1)*dx
 enddo
 u1(1:nx/5)=sin(1.6*x(1:nx/5))
 u2 = u1
 u3 = u1
 u4 = u1
 do k = 1,nt
   do i = 2,nx-1
     u2(i) = u1(i)-0.5*dt/dx*(F(u1(i+1))-F(u1(i-1)))
     u4(i) = u3(i)-0.5*dt/dx*(F(u3(i+1))-F(u3(i-1))) &
```

```
                +mu*dt/(dx**2)*(u3(i-1)-2.0*u3(i)+u3(i+1))
      enddo
      u1 = u2
      u3 = u4
    enddo
    z(1,:) = x
    z(2,:) = u2
    z(3,:) = u4
    z(4,1:nx/5)=sin(1.6*x(1:nx/5))
    open(7,file='Burgers.dat')
    write(7,91) (z(1,i),z(2,i),z(3,i),z(4,i),i=1,50)
91  format(4(2x,f11.6))
   end program Burgers

   real function F(x)
     real x
     F = 0.5*x*x
   end function F
```

9.8. Эллиптические уравнения

Эллиптическое уравнение в двумерной постановке можно записать в следующем виде:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0. \tag{9.83}$$

Уравнение (9.83) можно записать, используя оператор Лапласа:

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

Используя операторные обозначения можно записать основные эллиптические уравнения – Лапласа, Пуассона и Гельмгольца:

$$\triangle^2 u = 0, \quad \text{уравнение Лапласа,} \tag{9.84}$$

$$\triangle^2 u = f(x,y), \quad \text{уравнение Пуассона,} \tag{9.85}$$

$$\triangle^2 u + f(x,y)u = g(x,y), \quad \text{уравнение Гельмгольца.} \tag{9.86}$$

Введем в прямоугольной области $\Re = \{(x, y) : 0 \leqslant x \leqslant a,\ 0 \leqslant y \leqslant b\}$ прямоугольную сетку с постоянными шагами по x и y. Пусть на границах этой прямоугольной области заданы граничные условия $f(x, y)$ и $g(x, y)$.

9.8.1. Граничные условия

При решении эллиптических уравнений большое значение имеют граничные условия. Так как эти уравнения не зависят от времени, решение не зависит от начальных условий.

Граничные условия, которые задаются на границах области $\Re$ бывают трех видов – Дирихле (8.9), Неймана (8.10) и Робина (8.11). Граничное условие Дирихле задает постоянную температуру (для уравнения теплопроводности) на границах области.

Сложнее задать граничное условие Неймана. В этом случае, на границе $\Re$ задается производная по нормали к границе:

$$\frac{\partial u(x, y)}{\partial n}\Big|_{\Re} = \chi,$$

Рассмотрим случай, когда производная равна нулю

$$\frac{\partial u(x, y)}{\partial n}\Big|_{\Re} = 0.$$

Рассмотрим аппроксимацию граничного условия Неймана на правой границе $\Re$.

$$\frac{\partial u(x_{N,j})}{\partial x})\Big|_{\Re} = 0. \tag{9.87}$$

Тогда, уравнение Лапласа для точки $u(x_N, y_j)$ примет вид:

$$(u_{N+1,j} + u_{N-1,j} + u_{N,j+1} + u_{N,j-1} - 4u_{N,j}) = 0. \tag{9.88}$$

Найдем неизвестное значение $u_{N+1,j}$. Это значение находится за пределами расчетной области. Аппроксимируя найдем значение $u_{N+1,j}$:

$$u_{N+1,j} \approx \frac{u_{N+1,j} - u_{N-1,j}}{2\triangle x} = 0.$$

Отсюда следует, что с точностью $O(dx^2)$ $u_{N+1,j} = u_{N-1,j}$. Подставляя в уравнение (9.88) получим:

$$2u_{N-1,j} + u_{N,j+1} + u_{N,j-1} - 4u_{N,j} = 0, \quad \text{правая граница.} \tag{9.89}$$

Таким образом мы нашли выражение для аппроксимации граничного условия Неймана на правой границе. Рассуждая аналогичным образом, найдем выражения для граничных условий на остальных границах:

$$\begin{aligned} 2u_{N-1,j} + u_{N,j-1} + u_{N,j+1} - 4u_{N,j} &= 0, \quad \text{левая граница}, \\ 2u_{i,M-1} + u_{i-1,M} + u_{i+1,M} - 4u_{i,M} &= 0, \quad \text{верхняя граница}, \\ 2u_{i,2} + u_{i-1,1} + u_{i+1,1} - 4u_{i,1} &= 0, \quad \text{нижняя граница}. \end{aligned} \tag{9.90}$$

Проведем дискретизацию уравнений (9.84) – (9.86), для этого разделим область $\Re$ на прямоугольники dx, dy, так, чтобы $a = mxdx,\ b = mydy$. Если положить $dx = dy$ – тогда в области $\Re$ можно записать:

$$\nabla^2 u = \frac{u(x+dx,y)+u(x-dx,y)+u(x,y-dy)+u(x,y+dy)-4u(x,y)}{(dx)^2} + O((dx)^2). \tag{9.91}$$

Из (9.91) уравнение Лапласа можно представить в виде:

$$\frac{u(x+dx,y)+u(x-dx,y)+u(x,y-dy)+u(x,y+dy)-4u(x,y)}{(dx)^2} = 0. \tag{9.92}$$

Как и (9.91) оно имеет порядок точности $O(dx^2)$. Если обозначить $u_{i,j} = u(x_i, y_j)$ получим:

$$\nabla^2 u_{i,j} \approx \frac{c}{(dx)^2} = 0. \tag{9.93}$$

Это простая итерационная схема построена на пятиточечном шаблоне и называется методом Якоби. Умножая (9.93) на dx^2 получим формулу для численного решения уравнения Лапласа. Решение по формуле (9.93) находится методом простой итерации. Этот метод сходится достаточно медленно, поэтому для его ускорения используют метод Гаусса – Зейделя. Этот метод, как и в случае нахождения решения системы линейных уравнений использует уже найденные значения для ускорения сходимости. Если обозначить верхним индексом номер итерации, то уравнение (9.92) можно записать в виде:

$$u_{i,j}^{k+1} = 0.25 * (u_{i+1,j}^{k} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k} + u_{i,j-1}^{k+1}) \tag{9.94}$$

Представим методы Якоби и Гаусса – Зейделя в матричном виде. Для того, чтобы привести уравнения (9.92) и (9.94) к стандартному матричному виду вместо u будем использовать x. Таким образом, для того, чтобы решить уравнение

$$A \cdot x = b. \tag{9.95}$$

Представим A в виде:

$$A = L + D + U, \tag{9.96}$$

здесь L – нижняя треугольная матрица с нулевой диагональю, D – диагональная матрица, U – верхняя треугольная матрица с нулевой диагональю. Итерационный метод Якоби может быть представлен в виде:

$$D \cdot x^{(k+1)} = -(L + U) \cdot x^{(k)} + b. \tag{9.97}$$

Как известно (3.5), метод Якоби сходится, если матрица A является строго диагонально доминирующей. Найдем скорость сходимости методов Якоби и Гаусса – Зейделя. Очередная итерация метода Якоби:

$$x^{(k+1)} = -D^{-1}(L + U)x^{(k)} + b$$

Найдем собственные значения матрицы $D^{-1}(L + U)$. Скорость сходимости можно найти аналитически для каждого уравнения со своими граничными условиями и своей геометрией расчетной сетки. Возьмем, для примера, уравнение (9.84) на расчетной сетке размером MxM, с граничными условиями Дирихле на всех четырех сторонах квадратной расчетной области. Для больших значений M можно получить асимптотическое выражение:

$$q \approx 1 - \frac{\pi^2}{2M^2}. \tag{9.98}$$

Количество итераций, которое необходимо провести для того, чтобы получить ошибку не более чем 10^{-p} равно:

$$k \approx \frac{2pM^2 \ln 10}{\pi^2} \approx \frac{1}{2}pM^2. \tag{9.99}$$

Число итераций, необходимых для достижения требуемой точности пропорционально числу узлов сетки M^2. Отсюда следует, что метод Якоби представляет больше теоретический или методический интерес.

Для метода Гаусса – Зейделя соответствующая матрица имеет вид

$$(L + D)x^{(k+1)} = -Ux^{(k)} + b. \tag{9.100}$$

Так как матрица L находится слева от знака равенства, количество итераций, которые необходимо провести для получения необходимой точности:

$$k \approx \frac{pM^2 \ln 10}{\pi^2} \approx \frac{1}{4}pM^2. \tag{9.101}$$

Количество итераций по сравнению с методом Якоби уменьшилось в два раза, но все равно время расчета на больших сетках будет все еще слишком велико.

Упражнение. Напишите программы, реализующие методы Якоби и Гаусса – Зейделя.

9.8.2. Метод последовательной сверхрелаксации (SOR)

Для получения лучшего алгоритма было предложено использовать коррекцию величины $x^{(k+1)}$ на стадии итераций Гаусса – Зейделя для предсказания дальнейших коррекций. Решая уравнение (9.100) для $x^{(k+1)}$ путем добавления и вычитания $x^{(k)}$ из правой части уравнения получим:

$$x^{(k+1)} = x^{(k)} - (L+D)^{-1} \cdot [(L+D+U) \cdot x^{(k)} - b]. \tag{9.102}$$

Выражение в квадратных скобках есть остаточный член $\xi^{(k)}$, так что

$$x^{(k+1)} = x^{(k)} - (L+D)^{-1} \cdot \xi^{(k)}. \tag{9.103}$$

Теперь *суперкоррекцию* можно определить:

$$x^{(k+1)} = x^{(k)} - \omega(L+D)^{-1} \cdot \xi^{(k)}. \tag{9.104}$$

Здесь ω называется параметром последовательной сверхрелаксации. Для ω можно сделать следующие допущения:

- Значение ω находится в диапазоне $0 < \omega < 2$. Если $0 < \omega < 1$ говорят о нижней релаксации.
- Из некоторых математических соображений можно утверждать, что когда $1 < \omega < 2$ можно говорить о быстрой сходимости метода Гаусса – Зейделя.

Параметр ω можно найти из (9.105):

$$\omega_n = \begin{cases} 0, \quad \text{для} \quad k = 0, \\ \frac{1}{(1-\frac{\rho^2}{2})}, \ \text{для} \quad k = 1, \\ \frac{1}{(1-\frac{\rho^2 \omega_1}{4})}, \quad \text{для} \quad k = q = 2, \\ \frac{1}{(1-\frac{\rho^2 \omega_{q-1}}{4})} \quad \text{для} \quad k = q > 2. \end{cases} \tag{9.105}$$

Здесь $\rho = 1 - (\frac{\pi}{2(k+1)})^2$ - спектральный радиус.

Число итераций, которые необходимо совершить для получения точности 10^{-p} равно (9.100). Если сравнить уравнения (9.99) и (9.105), то можно увидеть, что метод последовательной сверхрелаксации требует порядка M итераций, в то время как метод Гаусса - Зейделя M^2.

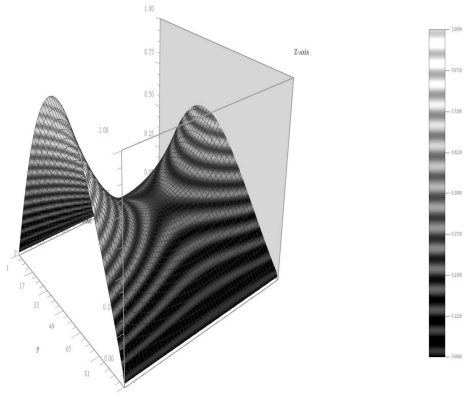

Рис. 9.15:

Пример 9.7 Методом последовательной сверхрелаксации решить уравнение Лапласа

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{9.106}$$

в прямоугольной области $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$ с **начальным условием**:

$$u(x,y)\big|_{t=0} = 0, \tag{9.107}$$

и **граничными условиями**:

$$\begin{aligned} u(x,0) &= \sin(\pi x), \\ u(x,1) &= \sin(\pi x)e^{-\pi}, \\ u(0,y) &= u(1,y) = 0. \end{aligned} \tag{9.108}$$

Аналитическое решение дается уравнением:

$$u(x,y) = \sin(\pi x)e^{-\pi y}. \tag{9.109}$$

Ниже приведена программа решение уравнения Лапласа методом SOR. Все пояснения к программе представлены в виде комментариев.

На рисунке 9.15 приведено численное решение уравнения (9.106) с начальным (9.107) и граничными (9.108) условиями. Решение найдено за num=5534 итерации, значение параметра сверхрелаксации к концу расчета равно $\omega = 1.99$.

Программа 9.7 Решение уравнения Лапласа методом SOR

```
    module ModSOR
      integer,parameter::nx=100, ny=100, maxIter=100000
      integer::i,j,num = 0
      real:: Pi = 3.141592653589793
      real,dimension(nx,ny)::u1,u2,u3,ue,UContrl,maxdiff
      real,dimension(nx)::   WL,WR,BL,BR,x
      real,dimension(ny)::   WT,WB,BT,BB,y
!  u1,u2 - массивы для хранения вычисленных значений на двух
 ! временных шагах.

 !  WL,WR - одномерные массивы для хранения весов
 !  на левой и правой границах
 !  WT,WB - одномерные массивы для хранения весов на верхней
 !  и нижней границах
 !  BL,BR - одномерные массивы для хранения граничных значений
 !  на левой и правой границах
 !  BT,BB - одномерные массивы для хранения граничных
 !  значений на верхней и нижней границах

      real:: dx,dy,dNx,dNy,SNx,SNy
!  dx,dy - шаги по x и y соответственно.

 !  SNx,SNy - нормированная длина сторон расчетной области по
 !  x и y соответственно.

!  dNx,dNy - нормированная длина ячеек по x и y соответственно.
     real::ro,omega0,omega1,omega2,omegan,TContrl,eps=1.0E-06
     contains
      subroutine boundary()
        real:: z(nx),z1(ny),dd
        dx = 1./nx
        dy = 1./ny
```

```
  dd = dx
 do  i = 1,nx
   x(i) = (i-1)*dx
   BT(i) = 0.0
   BB(i) = 0.0
 enddo
 do  j = 1,ny
   y(j) = (j-1)*dy
   BL(j) = sin(Pi*y(j))
   BR(j) = sin(Pi*y(j))*exp(-y(j))
 enddo
 z = BR
 z1 = BL
end subroutine boundary

subroutine FirstIterations()
  real:: z(nx),z1(ny),w(nx,ny)
  SNx = dx*nx
  SNy = dy*ny
  dNx = SNx/nx
  dNy = SNy/ny
 do  i = 1,nx
   WL(i) = (1.-i*dNx)
   WR(i) = i*dNx
 enddo
 do  j = 1,ny
   WT(j) = (1.-j*dNy)
   WB(j) = j*dNy
 enddo
  z = WL
  z1 = WB
do j = 1,ny
 do i = 1,nx
   u1(i,j) = 0.25*(WL(i)*BL(j)+WR(i)*BR(j)+WB(j)*BB(i)+ &
             WT(j)*BT(i))
 enddo
enddo
w=u1
end subroutine FirstIterations
```

```
      subroutine ExactSolution
         real,dimension(nx,ny):: z,zz
        do j = 1,ny
          do i = 1,nx
            ue(i,j) = sin(Pi*x(i))*exp(-x(i)*y(j))
          enddo
        enddo
         z = ue
         zz = ue-u1
      end subroutine ExactSolution
    end module ModSOR
    program MainSor
      use ModSOR
 ! Решение уравнения Лапласа методом SOR - последовательной
 ! сверхрелаксации
 ! u(x,0) = sin(Pi*x),          0<=x<=1
 ! u(x,1) = sin(Pi*x)exp(-x), 0<=x<=1
 ! u(0,y) = u(1,y) = 0 0<=y<=1
 ! u(x,y) = sin(Pi*x)*exp(-x*y) -точное решение
 ! Используется начальное приближение - осредненное значение
 ! по 4 границам
 ! Скаляр сверхрелаксации 0<omega<2 определяем по формуле
 ! omega(0) = 0
 ! omega(1) = 1/(1-ro**2/2), ro - спектральный радиус
 ! ro=1-(Pi/(2*(n+1))**2
 ! omega(2) = 1/(1-ro**2*omega(1)/4)
 ! omega(i>2) = 1/(1-omega(i-1)/4)
      call boundary()
      call FirstIterations()
      call ExactSolution()
      call SOR()
    end program MainSor
        subroutine Omega(om,key)
      use ModSOR
      real::om
!  ro - спектральный радиус
      ro = 1.-(Pi/(2.*(key+1)))**2
      omega0 = 0.
      omega1 = 1./(1.-0.5*ro*ro)
```

```
      omega2 = 1./(1.-0.25*ro*ro*omega1)
      omegan = 1./(1.-0.25*ro*ro*omega2)
    if( key > 2 ) then
       om = omegan
    elseif( key == 2 ) then
       om = omega2
    elseif( key == 1 ) then
      om = omega1
    else
      om = omega0
    endif
    end subroutine Omega
        subroutine SOR()
      use ModSOR
      real zz(nx,ny)
     do 11 k = 1,maxIter
       num = num+1
       u1(1,:)  = BL
       u1(nx,:) = BR
       u1(:,1)  = BB
       u1(:,ny) = BT
       u2 = u1
       u3 = u1
       omegan = 1./(1.-0.25*ro*ro*omegan)
      call Omega(omegan,k)
      do 3 j = 2,ny-1
       do 1 i = 2,nx-1
         u2(i,j) = 0.25*(u2(i-1,j)+u2(i,j-1)+u1(i+1,j)+ &
                   u1(i,j+1))
1     enddo
3    enddo
     do 7 j = 2,ny-1
       do 5 i = 2,nx-1
         u3(i,j) = omegan*u2(i,j)+(1.-omegan)*u1(i,j)
5     enddo
7    enddo
     UContrl = abs(u2-u3)  !**2
     maxdiff = 0.
     maxdiff = max(maxdiff,UContrl)
```

```
     dmax = maxval(maxdiff)
    if( dmax < eps ) exit
      u1 = u2
      u2 = u3
      zz = u3
      if( mod(num,10000) == 0 ) then
         call vGraph2(UContrl,nx,ny)
         write(*,*) 'num = ',num,'omegan = ',omegan, &
                    ' TContrl = ',dmax
      endif
      TContrl = 0.
      UContrl = 0.
11   enddo
     call vGraph2(u3,nx,ny)
     write(*,*) ' dmax = ',dmax, ' num = ',num
     write(*,*) 'num = ',num,'omegan = ',omegan, &
                ' dmax = ',dmax
    end subroutine SOR
         subroutine vGraph2(fun,nx,ny)
      use avdef
      use avviewer
      use dflib
      integer(4)::nvalues
      real(4)::fun(nx,ny)
      integer(4)::hv,status,nError
      character(1)::key
      character(av_max_label_len)::xLabel='y'
      call faglStartWatch(fun,status)
      write(*,*) 'Starting Array Viewer'
      call favStartViewer(hv,status)
      if( status /= 0 ) then
       call favGetErrorNo(hv,nError,status)
      if( nError /= 0) then
        write(*,*) "ArrayViewer reports error",nError
        stop
      endif
      endif
      call favSetArray(hv,fun,status)
      call favSetArrayName(hv,"Van de Pol Cycle",status)
```

```
      call favSetGraphType(hv,HeightPlot,status)
      call favSetUseAxisLabel(hv,x_axis,1,status)
      call favSetAxisLabel(hv,x_axis,xLabel,status)
      call favShowWindow(hv,av_true,status)
      write(*,*) "Press any key to close down the viewer"
      key = getcharqq()
!      call sleepqq(3000)
      call favEndViewer(hv,status)
      call faglEndWatch(fun,status)
    end
```

9.9. Упражнения

1. Провести классификацию следующих уравнений:

   1. $u_{xx} + xu_{tt} = 0$
   2. $u_t - uu_{xxx} = \sin(x)$
   3. $u_t - a^2 u_{xx} = 0$
   4. $4u_{xx} + 5u_{xt} + 2u_{tt} = 0$
   5. $u_t u_{xx} - 3x^3 uu_{xt} + 3u_x - u = 0$

2. Найти численное решение линейного уравнения Бюргерса

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = \mu\frac{\partial^2 u}{\partial x^2},$$

с начальным условием $u(x,0) = \sin(kx)$ и периодическими граничными условиями. Сравнить с точным решением

$$u(x,t) = e^{-k^2\mu t}\sin[k(x-ct)].$$

3. Найти решение уравнения Лапласа в прямоугольной области $0 \leqslant x \leqslant 1.5$, $0 \leqslant y \leqslant 1.5$, с нулевым начальным и следующими граничными условиями:

$$\begin{aligned} u(x,0) &= x^4, \quad u(x,1.5) = x^4 - 13.5x^2 + 5.0625, \\ u(0,y) &= y^4, \quad u(1.5,y) = y^4 - 13.5y^2 + 5.0625. \end{aligned}$$

Глава 10

Технологии параллельного программирования

10.1. Введение

Необходимость включения в данную монографию глав, посвященных построению параллельных методов и алгоритмов, вызвана тем, что параллельные ЭВМ получают все более широкое распространения. В последние годы наметилась тенденция перехода от одноядерных процессоров к многоядерным. Это вызвано как совершенствованием технологических процессов при выпуске процессоров, так и необходимостью дальнейшего роста производительности компьютеров. Цены на многоядерные процессоры снижаются, что делает их все более доступными. С другой стороны, постоянно усложняются прикладные задачи, которые требуют больше вычислительных мощностей. Как отмечается в [6] *параллелизм в архитектуре компьютеров – это не просто надолго, это навсегда.*

Для того, чтобы создавать эффективные параллельные алгоритмы, необходимо хотя бы в общих чертах знать, как устроены параллельные ЭВМ. Приведем одну из наиболее известных схем классификации компьютеров, предложенную М. Флинном. В классификации Флинна все компьютеры относятся к одному из четырех классов:

1. **SISD** Single Instruction – Single Data – один поток команд и один поток данных. Это обычный последовательный компьютер, в котором один оператор обрабатывает один элемент данных.

2. **SIMD** Single Instruction – Multiple Data – один поток команд и несколько потоков данных. В таких процессорах процессом вы-

числений управляет специальный элемент, контроллер, а вычисления выполняют процессорные элементы. Разновидностью системы SIMD являются веторные компьютеры.

3. **MISD** Multiple Instruction – Single Data – несколько потоков команд и один поток данных. Компьютеры такого класса обычно состоят из большого числа процессоров, которые объединены в так называемый систолический массив. В каждый момент времени каждый процессор выполняет одну операцию и передает ее соседнему процессору.

4. **MIMD** Multiple Instruction – Multiple Data несколько потоков команд и несколько потоков данных. Это самый распространенный класс компьютеров с параллельной архитектурой. В этой архитектуре каждый из процессоров обрабатывает какую - либо подзадачу общего потока команд.

Классификация Флинна до последнего времени была самой распространенной классификацией. Однако эта классификация не учитывает более детальную организацию компьютеров. На сегодняшний день существуют другие, более специализированные классификации, например Р. Хокни, Т. Фенга и так далее. Более подробно с ними можно ознакомится в монографии [6].

В связи с появлением параллельных ЭВМ возникла необходимость в создании принципов параллельного программирования. Традиционное, последовательное программирование, может служить отправной точкой при создании параллельной программы. При разработке параллельных программ необходимо дополнительно принимать во внимание то, каким образом подзадачи будут обмениваться между собой информацией, как равномерно загрузить имеющиеся процессоры, как избежать конфликта между процессорами и так далее.

10.2. Архитектура параллельных ЭВМ

Традиционная архитектура фон Неймана позволяет выполнять последовательно один оператор за другим. Напомним, что архитектура фон Неймана основана на следующих принципах:

- программа и данные, которые она обрабатывает, хранятся в оперативной памяти;

- компьютер состоит из следующих компонент:

    1. процессора, который в свою очередь состоит из АЛУ - арифметико – логического устройства и устройства управления;
    2. оперативной памяти (ОЗУ);
    3. внешней памяти;
    4. устойства ввода – вывода.

Существует несколько различных архитектур процессоров – CISC, RISC и VLIW. Архитектура CISC (Complete Instruction Set Computer) - компьютер с полным набором команд. В процессоре с такой архитектурой выполнение каждой команды занимает разное время. В таких процессорах не удается организовать высокоэффективную конвейерную обработку данных.

Процессоры с RISC архитектурой (Reduced Instruction Set Computer) это процессоры с сокращенным набором команд (тип архитектуры микропроцессора, ориентированный на быстрое и эффективное выполнение относительно небольшого набора встроенных команд). Все команды разбиваются на простые и сложные. В обычных вычислениях порядка 80% команд – простые, и около 20% сложные. Все простые команды выполняется за одно и то же время. Для сложных команд существуют специальные блоки, которые ускоряют их обработку. За счет этого достигается большая скорость выполнения операций, так как удается организовать конвейерную обработку данных. Основная идея конвейера заключается в следующем. Сложную операцию разбивают на несколько более простых, часть из которых может выполняться параллельно. В своей работе ковейер позволяет осуществлять различные подоперации на различных участках ковейера. Применение ковейеров позволяет существенно ускорить работу процессора. Таких конвейеров в процессоре может быть несколько. За счет RISC архитектуры и организации ковейеров удается добиться того, чтобы процессор обрабатывал команду за один или несколько тактов. Такие процессоры называются скалярными. Суперскалярные процессоры обеспечивают обработку нескольких команд за один такт. Это обеспечивается использованием нескольких конвейеров в процессоре.

VLIW (Very Large Instruction Word) процессоры – это процессоры, в которых формируется сверхдлинное командное слово, которое состоит из набора полей. Каждое поле отвечает за определенную операцию. Повышение быстродействия осуществляется за счет того, что на каждом

такте процессора выдается команда, которая содержит информацию о параллелизме всех операций.

Параллельные системы разделяются на системы с **общей памятью** и **локальной памятью**. Компьютеры с **общей памятью** имеют несколько процессоров, каждый из которых имеет доступ к общей памяти. Если у каждого процессора существует своя память, и другие процессоры не имеют к ней прямого доступа – то такие вычислительные системы называются компьютерами с **распределенной памятью**. Эта память "разделяется" между всеми процессорами.

Каждая из систем имеет свои достоинства и недостатки. Для систем с общей памятью не требуется организовывать передачу информации, так как она доступна всем процессорам. Но как только один процессор обращается к какому нибудь участку памяти, его нужно заблокировать для того, чтобы другие процессоры не смогли "испортить" этот участок памяти. Таких проблем нет в системах с распределенной памятью, но в этих системах необходимо организовать обмен информацией между процессорами.

К компьютерам с общей памятью относятся системы класса SMP (Symmetric Multi Processors). Все процессоры такой системы имеют абсолютно одинаковый доступ к памяти и ко всем другим ресурсам системы.

Другим примером построения многоядерных систем служит архитектура ccNUMA (cache coherent Non Uniform Memory Access) компьютер с неоднородным доступом к памяти и согласованным кэшем. В такой архитектуре многопроцессорная конфигурация составлена из RISC процессоров, у которых доступ к памяти неоднороден. Это означает, что время обращения к памяти своего ядра меньше, чем обращение к памяти удаленного ядра. Для программиста эта технология доступа к оперативной памяти почти не отличается от работы с архитектурой SMP, то есть программы, разработанные для SMP будут работать с ccNUMA также эффективно. Но это верно только в том случае, когда разница во времени доступа к локальной и удаленной памяти не очень велика и составляет $5 - 10\%$. Если это не так, и время доступа к удаленной памяти больше, программисту придется правильно распределить необходимые данные.

Архитектура SMP позволяет строить вычислительные системы, состоящие из не очень большого, до 64 ядер. Решения на архитектуре ccNUMA позволяет объединять более 256 ядер.

Следующим компонентом компьютера, который оказывает большое влияние на скорость вычислений, является оперативная память. Если

скорость выборки информации из ОЗУ не очень большая, то процессор большую часть времени будет простаивать, ожидая, когда он получит необходимые данные. Скорость работы самого процессора намного больше скорости обращения к памяти. Для повышения быстродействия ОЗУ, вводят специальную иерархию оперативной памяти. Самой быстрой памятью являются регистры, которые расположены непосредственно в процессоре. Это самая быстродействующая память, но и самая маленькая по объему. Более медленной, но большей по объему является кэш-память первого уровня. Затем следует кэш-память второго уровня, которая соответственно работает медленнее кэша первого уровня, но больше его по объему. Третьей ступенькой в иерархии идет оперативная память. Регистры и кэш память делится в свою очередь на память для команд и память для данных.

Существуют специальные алгоритмы выборки информации из оперативной памяти в кэш память первого и второго уровней. И если программист правильно проектирует программу, он должен учитывать эти алгоритмы выборки. Так, напрмер, двумерные массивы в C++ располагаются по строкам, а в Фортране – по столбцам. Если не учитывать это, то можно написать очень неэффективную программу, которая будет работать с большими многомерными массивами не в том порядке, как они располагаются в памяти. Это приведет к тому, что алгоритм выборки запишет в кэш память первый элемент массива, с которым мы работаем, и остальные элементы этой строки. Но если они располагаются по столбцам, а мы будем обращатся к первому элементу второй строки, то алгоритму выборки придется убрать из кэш-памяти первую строку и разместить вторую. Такая неэффективная работа приводит к существенному замедлению работы процессора.

Кроме этой проблемы, при работе на многопроцессорной системе возникает проблема когерентности кэша. Так, например, если нескольким ядрам одновременно понадобилось значение некоторой ячейки памяти, то это значение будет записано в кэш память каждого ядра. И каждое ядро будет работать со своей собственной копией данных этой ячейки, не учитывая изменений, которые делают в ней другие ядра. Таким образом, при проектировании параллельных программ необходимо синхронизировать кэши разных ядер.

Таким образом, быстродействие компьютера определяется скоростью работы его центрального процессора и временем доступа к оперативной памяти. Однако традиционные схемы организации процессора не обеспечивают достаточно большого быстродействия, необходимого для ре-

шения сложных задач математического моделирования. Повышение быстродействия процессоров можно добиваться с помощью:

- конвейерной обработки данных,
- использования процессоров с сокращенным набором команд RISC,
- использования суперскалярных процессоров,
- использования процессоров со сверхдлинным словом VLIW
- использования многопроцессорных, кластерных конфигураций.

В последнее время появился новый тип многопроцессорных ЭВМ - кластеры.

Определение 10.1 *Кластер это группа компьютеров, объединенных в локальную вычислительную сеть и способных работать в качестве единого вычислительного ресурса*

Существует разные варианты кластерных систем. *Кластеры рабочих станций* представляют собой совокупность рабочих станций, соединенных в локальную сеть. Такие сети создаются в лаборатории, на факультета или в институте. Кластер такого типа можно считать вычислительной системой с распределенной памятью и распределенным управлением. Одним из первым кластеров такого типа был проект "Beofulf", который возник в NASA 1994 году. Кластер был собран из 16 процессоров Intel 486DX4/100 МГц. Процессоры были соединены обычной сетью Ethernet. В 1998 году в Лос-Аламоской национальной лаборатории был создан кластер Avalon. Он был собран из процессоров Alpha21164A частотой 533 МГц и работал под управлением операционной системе Linux. С того было времени было создано много различных кластеров. В списке самых быстродействующих компьютеров мира Top500 более трех четвертей занимают кластерные системы.

Рассмотрим, почему кластерные решения получили столь широкое распространение. Кластерная система, при относительно невысокой стоимости, может обладать производительностью, сравнимой с производительностью суперкомпьютеров. Необходимое для работы параллельного кластера программное обеспечение — бесплатное, в том числе и операционная система типа Solaris или SuSe. Библиотека для построения параллельных программ MPI рапространяется бесплатно.

Но для построения хорошо работающего параллельного кластера, необходимо решить целый ряд проблем. Прежде всего, следует иметь

в виду, что довольно часто в сеть объединяются компьютеры различных фирм-производителей, имеющие разную архитектуру, работающие под управлением разных операционных систем, имеющих разные файловые системы и т. д. То есть возникает проблема совместимости. Кроме того, необходимо, чтобы локальная сеть объединяющая процессоры системы имела достаточно большую пропускную способность.

10.2.1. Топология вычислительных систем

Для обеспечения эффективной работы многопроцессорный системы необходимо проанализировать интенсивность и направленность информационных потоков. В зависимости от анализа этих данных выбирается та или иная топология соединения процессоров между собой. Существует много различных способов соединения процессоров.

- **Линейка** (linear array or farm) – такая система соединения процессоров, при которой все процессоры содединены последовательно. То есть каждый процессор, кроме первого и последнего, связан с предыдущим и последующим.

- **Кольцо** (ring) – кольцо получается из линейки соединением первого и последнего процессоров.

- **Полный граф** (completly connected graph or clique)– система, в которой каждая пара процессоров соединена линией связи.

- **Звезда** (star) – центральный процессор связан линиями связи со всеми процессорами.

- **Двух или трехмерная решетка** (mesh) – процессоры соединены в двумерную или пространственную решетку

- **Гиперкуб** (hypercube)-–данная топология представляет частный случай структуры решетки, когда по каждой размерности сетки имеется только два процессора.

Топология сети характеризуется несколькими параметрами.

- **Диаметр** – это максимальное расстояние между двумя процессорами сети. Оно характеризует максимально-необходимое время для передачи данных между процессорами.

- **Связность** (connectivity) – минимальное количество дуг, которое нужно удалить, для разделения сети передачи данных на две несвязные области.
- **Ширина бинарного деления** (bisection width) – минимальное количество дуг, которое удалить для разделения сети передачи данных на две несвязные области одинакового размера.
- **Стоимость**– общее количество линий передачи данных в многопроцессорной вычислительной системе.

Топология	**Диаметр**	**Ширина**	**Связн.**	**Стоим.**
Полный граф	1	$p^2/4$	(p-1)	p(p-1)/2
Звезда	2	1	1	(p-1)
Линейка	(p-1)	1	1	(p-1)
Кольцо	p/2	2	2	p
Решетка	$2\sqrt{p}/2$	$2\sqrt{p}$	4	2p
Гиперкуб	Log(p)	p/2	Log(p)	pLog(p)/2

10.3. Алгоритмы параллельного программирования

Определение 10.2 ***Алгоритмом*** *называется конечная совокупность инструкций, которая приводит к решению ряда однотипных задач.*

Традиционно считалось, что существует три типа алгоритмов:

1. **последовательный**;
2. **разветвляющийся**;
3. **циклический**.

С появлением многопроцессорных ЭВМ к этим трем типам добавился четвертый тип алгоритма – **параллельный**. Конечно, в реальных задачах, тип алгоритма чаще всего смешанный. При проектировании последовательного алгоритма мы всегда должны получить один и тот же результат для одного и того же набора начальных данных. Эта однозначность результата определяется тем, что в каждый момент времени компьютер выполняет только одну команду.

На многопроцессорных компьютерах такой однозначности в общем случае не существует. Это происходит потому, что в каждый момент

времени каждый процессор обрабатывает свою команду, и если не предпринять дополнительных действий, результаты вычислений могут различаться. Эта неоднозначность возникает из-за того, что часть операций на одном компьютере зависит от результатов, полученных на другом компьютере. Отсюда возникает необходимость в синхронизации вычислений.

Такую синхронизацию легче осуществить, если представить параллельный алгоритм в виде специальной диаграммы – *графа информационных зависимостей*. Эта диаграмма описывает последовательность выполнения операций, их взаимную зависимость и возможность выполнять часть операций одновременно. В этом информационном графе вершинами являются операции или блоки операций, а дугами - каналы обмена информацией. В последовательной модели граф информационных зависимостей простой, так как все операции выполняются одна за другой.

В модели параллельного программирования часть операций или блоков операций может выполняться параллельно и между ними возникает необходимость обмена информацией. Программа получается более сложной и трудоемкой. Программисту необходимо управлять распределением подзадач между процессорами, организовывать пересылку информации, обеспечивать синхронизацию работы отдельных потоков.

Можно описать алгоритм с необходимо степенью детализации. На самом нижнем уровне компьютерный алгоритм представляет собой последовательность машинных команд. На более высоком уровне алгоритм представляется в виде ассемблерного кода. Еще выше находится описание программы на языке высокого уровня. Графически каждый из этих уровней можно представить в виде блок схемы, или информационного графа. Вершинами графа или узлами являются команды, а дугами информационные потоки.

Множество операций, выполняемые в исследуемом алгоритме решения вычислительной задачи, и существующие между операциями информационные зависимости могут быть представлены в виде ациклического ориентированного графа G=(V,R). $V = \{1, ...|V|\}$ -– множество вершин графа, представляющих выполняемые операции алгоритма, R – множество дуг графа; дуга r(i,j) принадлежит графу только если операция j использует результат выполнения операции i. Вершины без входных дуг могут использоваться для задания операций ввода, а вершины без выходных дуг -– для операций вывода.

V -–множество вершин графа без вершин ввода,

d(G) -– диаметр графа (длина максимального пути).

Для разработки параллельного алгоритма необходимо выделить части программы, который могут работать параллельно. Существуют четыре основных способа разбиения задачи на независимые подзадачи. Этот процесс называется декомпозицией.

1. Распараллеливание на уровне отдельных задач. Каждую задачу, со своим набором начальных данных, рассчитывают на отдельном процессоре, независимо от других задач. Это самый крупнозернистый алгоритм распараллеливания.

2. Геометрическая декомпозиция. Исходную задачу разбивают исходя из геометрических соображений на отдельные части, каждую из которых можно вычислять на отдельном процессоре. Например, если мы решаем задачу математической физики в двумерной или трехмерной постановке. Исходную сетку мы геометрически разбиваем на прямоугольники или параллелипипеды, каждый из которых можно рассчитывать на отдельном процессоре. Необходимо только правильно организовать пересылку информации на границах между отдельными частями сетки. Это среднезернистый алгоритм.

3. Распараллеливание на уровне отдельных блоков программы. Это распараллеливание на уровне циклов или других блоков программы. Такое разбиение называется мелкозернистым.

4. Распараллеливание на уровне отдельных операторов программы. Например, операцию сложения или скалярного умножения двух векторов, можно покомпонентно выполнять на отдельных процессорах, а затем результат объединить.

После завершения этапа декомпозиции задачи на независимые подзадачи необходимо спланировать операции обмена информацией между подзадачами. К этому этапу относится промежуточный обмен информацией, принятие глобальных решений, общих для всей задачи (например выбор длины шага из условия Куранта – Фридрихса – Леви), условие завершения задачи и так далее. При этом необходимо определить способы и пути передачи информации.

В целях повышения скорости расчета может возникнуть необходимость укрупнить подзадачи, то есть объединить часть их них. Это производится с целью уменьшить затраты времени на обмен информацией. После этапа укрупнения необходимо разместить получившиеся блоки

подзадач на процессоры. При этом необходимо сбалансировать нагрузку на процессоры так, чтобы все они работали максимально равномерно и эффективно.

10.3.1. Планирование вычислений

Повышение количества процессоров ведет к росту его пиковой производительности, но и одновременно к увеличению разрыва между пиковой и реальной производительностью.

Определение 10.3 *Пиковой производительностью называется максимальное количество операций, которое может быть выполнено вычислительной системой за единицу времени*

Определение 10.4 *Реальной производительностью называется реальное количество операций, которое может быть выполнено вычислительной системой за единицу времени*

Снижение реальной производительности происходит из-за различных причин. Иногда сама структура алгоритма такова, что в принципе невозможно достичь высокой производительности (смотри закон Амдала). Иногда задача плохо запрограммирована и не может использовать всех преимуществ многопроцессорной архитектуры. Повысить производительность в этом случае можно, но для этого необходимо иметь хорошую операционную систему, компиляторы, поддержку особенностей работы с оперативной памятью, повышать степень распараллеливания и улучшать скорость передачи информации между процессорами.

При проектировании и создании параллельного алгоритма необходимо предусмотреть, какими методами необходимо пользоваться при решении конкретной задачи. Рассмотрим простой пример, который иллюстрирует проблему планирования вычислений. Он называется "Стена Фокса" . При постройке стены несколькими каменщиками можно найти несколько вариантов распараллеливания работ.

1. **Конвейерное решение.** Вся работа делится между рабочими по горизонтальным участкам. Каждый каменщик строит только один горизонтальный ряд стены. Эффективность такого распараллеливания меньше 100%, так как верхние ряды можно укладывать только после того, как уложены нижние. Хорошая эффективность достигается только при достаточной длине стены.

2. **Геометрическое решение.** Разделение работы на вертикальные секции. Каждому каменщику отводится своя секция. В этом слу-

чае все рабочие могут начинать работу одновременно, но возникает проблема синхронизации. Существуют затраты на обмен информацией и синхронизацию. Кроме того необходимо обеспечить баланс нагрузок, связанный с неодинаковой производительность работы камещиков.

3. **Коллективное решение.** В этом случае кирпичи и цемент не выдаются каждому каменщику, а хранятся в одном месте. Каждый рабочий берет необходимые материалы, переносит их к месту работы и кладет в доступную позицию стены. В данном варианте возникает неэффективность в начале и в конце работы.

Несмотря на простоту примера, настоящие проблемы, возникающие при распараллеливании научных задач очень похожи на поблему со стеной Фокса. Кроме неэффективности, при распараллеливании возникают еще две проблемы: неопределенность и тупик.

1. **Неопределенность.** Рассмотрим следующую ситуацию. Пусть у нас есть трехпроцессорная система. Программа устроена так, что третий процессор принимает первое пришедшее значение, присваивает его переменной **a**, а второе пришедшее значение присваивает переменной **b**. Затем он вычисляет $c = 100 * a + b/10$. Пусть, для примера первый процессор посылает число 100, а второй 10. Так как неизвестно, какой процессор - первый или второй завершит работу раньше результат будет различным либо 10001, либо 1010.

2. **Тупик.** Такая ситуация возникает, когда один процесс ждет информацию от другого процесса, а он, в свою очередь, тоже ждет сообщения от первого процессора.

При проектировании параллельных вычислений может возникнуть **недетерминированная ситуация.** Она хорошо иллюстрируется примером, который предложил Дейкстра. Это так называемая задача "обедающие философы." В Оксфорде собираются 5 философов для обсуждения проблем. Однако даже философ иногда хочет кушать, поэтому в обеденной комнате накрывают, на котором есть пять тарелок, пять вилок и миска спагетти, которая наполняется по мере необходимости. Философ подходит к столу, берет две вилки, накладывает себе спагетти, кладет одну вилку на стол и начинает обедать. Все хорошо до тех пор, пока одновременно не придут обедать все пять философов, возьмут по одной вилке и не смогут наполнить себе тарелку.

При проектировании коммуникаций между процессорами необходимо корректно определить структуру каналов связи и сообщений, которыми будут обмениваться процессы между собой. Коммуникации бывают следующих типов:

1. **Локальные.** Каждая задача связана с небольшим количеством подзадач.

2. **Глобальные.** Каждая задача связана с большим количеством подзадач.

3. **Структурированные.** Каждая задача и связанные с ней подзадачи образуют регулярную структуру.

4. **Неструктурированные.** Каждая задача и связанные с ней подзадачи образуют произвольный граф.

5. **Статические.** Схема коммуникаций не меняется во время выполнения задачи.

6. **Динамические.** Схема коммуникаций динамически меняется во время выполнения задачи.

7. **Синхронные.** Отправитель и получатель синхронизируют обмен информацией.

8. **Асинхронные.** Обмен данными не координируется.

Для эффективной работе на многопроцессорной системе необходимо определить на скольких процессорах будут выполняться подзадачи. Критерий эффективности – минимизация времени выполнения задач. Стратегия размещения задач на процессорах представляет собой компромисс между требованиями максимальной независимости выполняющихся задач и учетом межпроцессорных сообщений. Существует несколько способов планирования вычислений. Одним из самых простых и эффективных является метод **хозяин – работник** (**Master – Slave**). В этом методе распределения нагрузки главная задача распределяет одинаковые задания между имеющимися процессорами. Каждая подчиненная задача получает от главной задачи задание и исходные данные для нее, обрабатывает эти данные и возвращает результат.

Другим способом планирования вычислений является иерархическая схема распределения задач. В этом методе возникает несколько уровней

иерархии, на каждом из которых есть главная задача и подчиненные задачи. Таким образом, на каждом уровне иерархии возникает метод **хозяин – работник**. Главная задача управляет "главными подзадачами" на каждом подуровне иерархии.

Третьим методом организации вычислений является децентрализованная схема. В таких схемах вычислений главная задача отсутствует. Задачи обмениваются необходимой информацией по мере возникновения необходимости.

10.4. Технология параллельного программирования

Для полного использования архитектуры многопроцессорной системы необходимо решить ряд задач.

- Необходимо найти в программе параллельные ветви, которые можно выполнять независимо.
- Распределить задачи и данные между процессорами.
- Согласовать распределение данных с параллелизмом задачи.

Параллельные ветви в программе можно найти, используя информационный граф программы. Но, даже для сравнительно простой задачи, сделать это непросто. Рассмотрим фрагмент программы на Фортране [6] :

```
do i =1,n
  do j = 1,n
    u(i+j)=u(2*n-i-j+1)*q+p
  enddo
enddo
```

Информационный граф этого фрагмента программы очень запутан. Из него непонятно, какие итерации можно выполнять независимо и можно ли вообще распараллелить этот фрагмент. Очень трудно догадаться, что этот фрагмент программы можно преобразовать к эквивалентной форме, в которой оба внутренних цикла будут параллельными.

```
  do i =1,n
    do 20 j = 1,n-i
20     u(i+j)=u(2*n-i-j+1)*q+p
    do 30 j = n-i+1,n
30     u(i+j)=u(2*n-i-j+1)*q+p
  enddo
```

Если первый фрагмент выполнить на суперЭВМ CRAY C90, то его производительность составит всего 20 MFlops, при пиковой производительности 1 GFlops. На втором фрагменте производительность составит 420 MFlops. Также не совсем понятно, можно ли распараллелить следующий фрагмент программы.

```
  do i =1,n
    u(i) = func(u,i)
  enddo
```

Если в теле функции используются элементы вида u(i-1) – то итерации зависимы, поэтому компилятор сгенерирует последовательный код.

В настоящее время используется несколько языков высокого уровня, позволяющих писать программы для многопроцессорных систем. Можно указать несколько подходов к созданию таких языков и компиляторов для них. Это:

1. традиционный язык программирования + новые конструкции распараллеливания.

2. традиционный язык + директивы обмена информацией между параллельными процессами.

3. специальные языки программирования.

Примером первой технологии является язык Фортран 95, в который встроены параллельные операторы, HPF (High Performance Fortran), OpenMP. К второй технологии относится библиотека MPI (Message Passing Interface). К третьей технологии относятся языки программирования OKKAM, ADA.

Определим понятие ускорения программы при ее выполнении на многопроцессорной ЭВМ.

Определение 10.5 *Ускорением будем называть увеличение производительности программы при ее выполнении на вычислительной системе, состоящей из p процессоров.*

Теоретическим пределом ускорения является линейный рост производительности при увеличении числа процессоров.

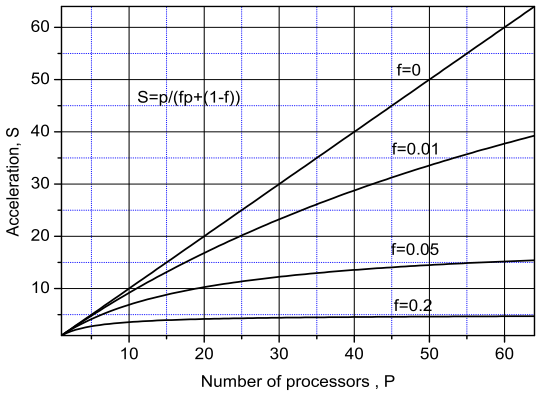

Рис. 10.1:

Определение 10.6 *Степенью параллелизма алгоритма называется число операций, которые можно выполнять параллельно.*

Примерами почти полностью параллельных алгоритмов является алгоритм сложения векторов размерности n. Если количество процессоров в системе больше или равняется n операции сложения компонент векторов можно выполнять параллельно. Другой задачей, которая хорошо распараллеливается, является задача вычисления числа π. Таким образом можно сказать, что *степень параллелизма зависит не от числа процессоров, а от самой задачи.*

Закон Амдала. Пусть наш алгоритм состоит из последовательной f и параллельной 1-f частей. Будем считать, что их сумма равна 1. Пусть p - количество одинаковых процессоров. Тогда максимально возможное ускорение алгоритма равно:

$$S = \frac{p}{fp + (1 - f)} \tag{10.1}$$

Для систем с общей памятью последовательную часть образуют операторы, которые выполняет главная нить программы. Для систем с локальной памятью последовательная часть программы образуется за счет операторов, выполнение которых дублируется всеми процессорами системы. Теоретически оценить последовательную часть программы очень

сложно, практически единственным средством оценки является запуск программы на многопроцессорных системах с различным числом процессоров. Таким образом, закон Амдала позволяет оценить максимальное количество процессоров, на которых с достаточной эффективностью будет выполняться параллельная программа.

10.4.1. Языки и методы параллельного программирования

Многопроцессорные ЭВМ, также как и обычные последовательные компьютеры могут выполнять только программы, написанные на машинном языке. Иногда, при создании эффективных математических библиотек, разработчики пишут программы в машинных кодах или на ассемблере. Программирование в машинных кодах или на ассемблере требует чрезвычайно высокой профессиональной подготовки программиста и трубует большого количества времени. Поэтому большинство программистов используют языки высокого уровня. В последнее время большое распространение получили проблемно-ориентированные языки, которые позволяют писать программы в профессиональных терминах той или иной предметной области. Для таких проблемно-ориентированных языков программирования необходимо создавать сложные компиляторы, которые преобразуют программу с языка высокого уровня в машинный код. Эффективность таких компиляторов и создаваемых ими кодов достаточно низкая.

Для быстрого создания новых параллельных программ хорошей идеей была бы возможность использовать традиционные последовательные языки программирования. При использовании хорошо ивестных языков проще начинать создавать новые параллельные программы. Можно пользоваться накопленными программами и библиотекеками. Все проблемы, связанные с распараллеливанием в этом случае, ложаться на компиляторы и операционные системы. Однако создание эффективных компиляторов для многопроцессорных систем является очень сложной задачей. Для систем с распределенной памятью необходимо, чтобы компилятор нашел в программе участки кода, которые можно выполнять параллельно. Затем необходимо распределить эти параллельные ветви по доступным процессорам. И далее координировать работу параллельных участков программы и организовывать обмен информацией между этими фрагментами. Анализ проблем, которые возникают при попытке создания компиляторов для автоматического распаралелливания пока-

зывает, что создать такой компилятор чрезвычайно трудно.

Рассмотрим те подходы к созданию новых технологий построения языков программирования, которые мы рассматривали в предыдущем параграфе. Примеров первого подхода - добавления в традиционный язык программирования новых конструкций распараллеливания - High Performance Fortran (HPF). В традиционные операторы Fortran были внесены специальные директивы, которые позволяли создавать параллельные программы. Однако эти конструкции были достаточно сложными, что не позволило широко использовать High Performance Fortran при программировании на многопроцессорных ЭВМ. Альтернативой HPF стало введение параллельных операторов в язык Fortran 95.

10.4.2. OpenMP

Для систем с общей памятью самой распространенной системой программирования стала технология программирования OpenMP. Программирование с использованием OpenMP заключается во введение в обычную последовательную программу специальных директив и процедур. OpenMP используется при разработке параллельных программ на FORTRAN 77, 90 и 95, а также для C и C++. Реализации OpenMP существуют для почти всех операционных систем и он включен в состав MS Visual Studio 2005 и Sun Studio 12 для операционной системы Solaris. Инициализаторами создания OpenMP выступили известные фирмы-создатели многопроцессорных ЭВМ и программного обеспечения для них: IBM, HP, DEC, SGI/CRAY, Portland Group и другие.

Все примеры, рассмотренные ниже были построены в операционных системах Windows XP (SP2), Windows Vista, Solaris 10. Были использованы компиляторы фирмы INTEL Fortran 9.1 (для Microsoft Visual Studio 2005), и Fortran для Sun Studio 12 фирмы Sun. Программы тестировались на четырехядерном компьютере с процессором Intel Core 2 Quad Q6600 (тактовая частота 2.4 ГГц), и двухядерном ноутбуке с процессором Intel Core 2 Duo Santa Rose T7500 (тактовая частота 2.2 ГГц).

Начнем с традиционной программы, которая печатает сообщение "Привет многопроцессорный мир от нити = ". Ниже приведен результат работы такой программы. Отметим, что нити напечатали свое сообщение в произвольном порядке. Это подтверждает то, что перевод английского термина concurrent на русский язык как параллельный не совсем полно отражает суть дела. Помимо значения параллельный термин concurrent имеет смысл соперничающий, конкурирующий.

```
    PROGRAM HELLO
      USE OMP_LIB
      INTEGER NUMBER
      CALL OMP_SET_DYNAMIC (.FALSE.)
      CALL OMP_SET_NUM_THREADS (10)
!$OMP PARALLEL PRIVATE (NUMBER)
! Obtain thread ID.
      NUMBER = OMP_GET_THREAD_NUM()
! Print thread ID.
      WRITE(*,*) 'Hello multiprocessing World from thread = ',
      NUMBER
!$OMP END PARALLEL
   END
```

```
 Hello multiprocessing World from thread =           2
 Hello multiprocessing World from thread =           1
 Hello multiprocessing World from thread =           7
 Hello multiprocessing World from thread =           6
 Hello multiprocessing World from thread =           0
 Hello multiprocessing World from thread =           9
 Hello multiprocessing World from thread =           8
 Hello multiprocessing World from thread =           5
 Hello multiprocessing World from thread =           4
 Hello multiprocessing World from thread =           3
```

Основные преимущества OpenMP заключаются в следующем:

1. OpenMP позволяет быстро и эффективно распараллелить большие программы.

2. Предоставляет эффективный и гибкий механизм для контроля хода выполнения параллельной программы.

3. В последовательной программе директивы OpenMP игнорируются.

Директивы OpenMP начинаются с символа коментария и специального префикса !$OMP. Если на компьютере не установлен OpenMP, компилятор проигнорирует эту директиву. Если OpenMP установлен, эта директива будет выполнена. С точки зрения программиста, распараллеливание с использованием OpenMP, представляет собой достаточно простую задачу. Возьмем классический пример вычисления числа π

Пример 10.1

```
program P1
  implicit none
  integer,parameter:: num=100000
  integer:: k
  real(8)::Pi,sum=0.0_8,fun,x,a,z
  fun(x)=4.0_8/(1.0_8+x*x)
  z = 1.0_8/num
 do k = 1,num
   x = z*k
   sum = sum+fun(x)
 enddo
 pi = sum*z
 write(*,*) 'Pi is equal ',pi
end program P1
```

Построить параллельный алгоритм этой программы, используя OpenMP очень просто. Достаточно добавить несколько операторов:

Пример 10.2

```
program P2
  use omp_lib
  implicit none
  integer,parameter:: num=100000
  integer:: k
  real(8)::Pi,sum=0.0_8,fun,x,a,z
  fun(x)=4.0_8/(1.0_8+x*x)
  z = 1.0_8/num
!$omp parallel
!$omp do private(x), shared(z)

!$omp & reduction(+:sum)
 do k = 1,num
   x = z*k
   sum = sum+fun(x)
 enddo
!$omp end do
!$omp end parallel
 pi = sum*z
 write(*,*) 'Pi is equal ',pi
end program P2
```

В данной программе, при ее старте создается основная, последовательная нить программы, которая имеет нулевой номер. Директива OpenMP

```
!$omp parallel do private(x), shared(z)
```

создает параллельные нити. Каждая нить получает свой номер и работает над своей частью программы. После завершения параллельной области автоматически создается точка синхронизации, в которой основная нить дожидается завершения всех параллельных нитей и дальше продолжает работать мастер-нить.

Программный интерфейс приложения OpenMP позволяет создавать направляемое пользователем распараллеливание, в котором пользователь явно определяет действия, которые будут предприняты компилятором и системой во время выполнения, чтобы выполнить программу параллельно. Программный интерфейс OpenMP не обязан проверять все зависимости, конфликты, тупики, и другие проблемы, возникающие при распараллеливании.

Пользователь, который использует OpenMP должен разбить программу на последовательные и параллельные участки. При запуске программы создается основной процесс или нить. Этот основной процесс порождает параллельные нити, которые выполняются на нескольких процессорах. OpenMP работает на компьютерах с SMP архитектурой, на которых существует поддержка заданий, выполняющихся на параллельных процессорах. Поэтому подобные архитектуры гораздо эффективней классических параллельных UNIX-процессов.

Приведем основную терминологию, встречающуюся при программировании с использованием технологии OpenMP.

- **базовый язык** - язык программирования, поддерживающий OpenMP.
- **базовая программа** - программа, написанная на базовом языке.
- **директива OpenMP** - оператор OpenMP, который предписывает выполгить то или иное действие.
- **программа OpenMP** - программа, состоящая из исходной программы на основном языке программирования, использующая директивы OpenMP.

- **блочная директива** - директива, состоящая из заголовка, тела (операторов программы) и окончания директивы (аналогично блочным директивам Фортрана).
- **последовательная часть программы** - часть программы вне параллельной области.
- **вложенный блок** - часть программы, которая динамически выполняется внутри другого блока.
- **активный параллельный блок** - блок параллельного оператора **if**, если условие .true.
- **неактивный параллельный блок** - блок параллельного оператора **if**, если условие .false.
- **начальная нить** - нить, которая выполняет начальную и последовательную часть программы.
- **главная нить** - нить, которая создает параллельную часть программы и создает бригаду параллельных нитей.
- **бригада** - одна или более нитей, участвующих в выполнении параллельной части программы.
- **барьер** - точка синхронизации всех нитей бригады. После точки синхронизации не может выполнится ни один оператор, пока не выполнятся все нити бригады.

Терминология описания данных при программировании с использованием технологии OpenMP.

- **переменная** - имя объекта, значение которого определяется и переопределяется в процессе выполнения программы.
- **закрытая переменная** - переменная, которая в каждой нити имеет одно имя, но которая доступна только внутри своей нити.
- **открытая переменная** - переменная, которая доступна всем нитям бригады.
- **глобальная переменная** - переменная, которая существует все время работы программы и доступна всем нитям.

- **поддержка n уровней параллелизма** - в активной параллельной области существует n-1 активная нить. Счет нитей начинается с нуля.

- **поддержка OpenMP** - поддержка по крайней мере одного уровня параллелизма.

Основным отличительным признаком директивы OpenMP является наличие следующего префикса:

!$*omp*.

Этот префикс может появиться в любой позиции строки. После

!$*omp*

записывается директива OpenMP. Как можно видеть, директива OpenMP начинается с восклицательного знака, который в Фортране служит символом комментария. Таким образом, если на компьютере не установлен OpenMP, то такие директивы будут игнорироваться, если OpenMP установлен они будут распознаваться как соответствующие метакоманды.

Все переменные программы делятся на общие (SHARED) и локальные (PRIVATE). Все общие переменные доступны всем нитям и существуют в единственном экземпляре. Локальные переменные создаются в каждой нити и имеют в них одно и то же имя. Время жизни локальных переменных совпадает с временем жизни параллельных нитей.

Основной директивой, которая служит для создания параллельной части программы служит конструкция:

```
!$omp parallel [опция[,опция]...]
   блок операторов
!$omp end parallel
```

Здесь опция может быть одним из следующих:

- **if**(скалярное выражение)

- **private**(список)

- **shared**(список)

- **firstprivate**(список)

- **lastprivate**(список)

- **default(shared|private|none)**

- **reduction**({знак операции|имя встроенной процедуры}:список)
- **threadprivate**(целое скалярное выражение)

Рассмотрим эти опции. Они позволяют управлять данными во время выполнения параллельных операций. Опция

`private(список)`

позволяет задать приватные переменные. При этом, при входе в параллельную область, для каждой нити бригады нитей создается локальные копии переменных, описанных в списке. Начальные значения этих переменных не определены и после окончания параллельной области эти данные не доступны другим параллельным областям.

`shared(список)`

В списке данной опции задаются переменные, которые должны быть общими для всех параллельных областей.

`firstprivate(список)`

Эта опция позволяет задать начальные данные всем приватным переменным, входящим в список. Начальные значения будут взяты из исходных переменных, которые у них были до входа в параллельную область.

`lastprivate(список)`

Опция **lastprivate** позволяет обновить значения приватных переменных после выполнения параллельной области.

```
!$omp parallel
!$omp do lastprivate(k)
  do k = 1, numX
    u(k) = v(k)-3.1415*sin(w(k))
  enddo
!$omp end parallel
```

В приведенном выше примере, значение переменной цикла k будет равно numX+1, то есть таким же, как и в обычном, непараллельном цикле.

`default(shared|private|none)`

Опция **default** позволяет определить те переменные, которые не были заданы в списках **private** и **shared**. То есть все переменные, которые не имеют своего атрибута будут по умолчанию либо **private** либо **shared** соответственно. Если задать опцию **none**, то пользователь обязательно должен будет объявить все переменные либо с атрибутом **private** либо **shared**.

```
reduction(\{знак операции|имя встроенной процедуры\}:список)
```

Операция **reduction** будет выполнена над всеми локальными копиями переменной во всех параллельных нитях и результат будет присвоен исходной переменной. В качестве знака операции можно указать основные арифметические операторы +, -, *,. Кроме того можно воспользоваться логическими операциями **.OR., .AND., .EQV., .NEQV.** или встроенными функциями **MIN, MAX, IOR, IEOR, IAND**. Например:

```
!$omp parallel do default(private) reduction (+: U,V)
   do k = 1,numX
     call work(U1, V1)
       U = U + U1
       V = V + V1
   enddo
```

Параллельная часть программы начинается после директивы

```
!$omp parallel.
```

При этом создается набор (бригада) нитей, которая выполняет параллельую часть программы. Нить, которая создала параллельную часть программы, называется мастер-нитью и имеет нулевой номер. Количество нитей в бригаде является константой во все время выполнения параллельной части.

Блок операторов в параллельной области выполняется каждой нитью, хотя каждая нить может выполнять свою часть операторов. После завершения параллельной области создается точка синхронизации и только мастер-нить продолжает выполнение программы вне параллельной области. Если внутри нити одной параллельной секции появляется другая параллельная секция, создается новая бригада нитей, у которой мастер-нитью становится порождающая нить программы.

Во время выполнения параллельной задачи можно получить информацию о количестве нитей и получить номер конкретной нити.

```
  subroutine omp_set_num_threads(N) устанавливает число нитей,
     равное N.
  integer function omp_get_num_threads() возвращает число нитей.
  integer function omp_get_thread_num() возвращает номер "теку-
  щей"
     нити (из которой произошел вызов функции)
```

В приведенном ниже примере создается параллельная область с числом нитей равном 10.

Пример 10.3

```
program P3
  use omp_lib
  call omp_set_dynamic(.true.)
!$omp parallel num_threads(10)
  блок операторов
 !$omp end parallel
end program P3
```

В следующем примере директива **parallel** используется для построения параллельной программы. Каждая нить в параллельной области получает часть глобального массива x и обрабатывает эту часть массива. Получает число(номер) нити, в пределах команды, в диапазоне от нуля до **OMP_GET_NUM_THREADS** минус один [31].

Пример 10.4

```
    program P4
      real(4):: array(100000), dx=0.1
      call sub(array, 100000)
    end program p4

    subroutine subdomain(x, istart, ipoints)
      integer istart, ipoints
      real x(*)
      integer i
    do 100 i=1,ipoints
      x(istart+i) = sin(i*dx)
100 continue
    end subroutine subdomain

    subroutine sub(x, npoints)
```

```
      use omp_lib
      real x(*)
      integer npoints
      integer iam, nt, ipoints, istart
      open(7,file='omp.txt')
!$omp parallel default(private) shared(x,npoints)
      iam = omp_get_thread_num()
      nt = omp_get_num_threads()
      ipoints = npoints/nt
      istart = iam * ipoints
      write(7,91) iam, nt, ipoints, istart
     if (iam .eq. nt-1) then
       ipoints = npoints - istart
     endif
      call subdomain(x,istart,ipoints)
 !$omp end parallel
 91 format('iam = ',i3,' nt = ',i5,'  ipoints =&
',i7,' istart =  ',i7)
    end subroutine sub
```

Результаты работы данной программы представлены ниже. Этот результат был получен на 4 - х ядерном компьютере с процессором Intel Core 2 Quad Q6600. В параллельной секции нити печатали свои данные в произвольном порядке, поэтому сначала идет номер 1 затем 0 и т.д.

```
 iam =   1 nt = 4  ipoints =   25000 istart =   25000
 iam =   0 nt = 4  ipoints =   25000 istart =       0
 iam =   2 nt = 4  ipoints =   25000 istart =   50000
 iam =   3 nt = 4  ipoints =   25000 istart =   75000
```

На двухядерном компьютере были получены следующие результаты:

```
 iam =   0 nt = 2  ipoints =   50000 istart =      0
 iam =   1 nt = 2  ipoints =   50000 istart =  50000
```

В программе P2 директива **OMP_GET_THREAD_NUM()** определяет номер нити, а директива **OMP_GET_NUM_THREADS()** находит общее количество нитей в параллельной секции программы.

Создание цикла. Цикл в OpenMP задается с помощью директивы:

```
!$omp do [опция[[,] опция] ... ]
    do-loop
[!$omp end do [nowait]]
```

Опция может быть одной из следующих:

- private(список)
- firstprivate(список)
- lastprivate(список)
- reduction(operator|intrinsic_procedure_name:список)
- ordered
- schedule(kind[, chunk_size])

Если директива **!$omp end do** не задана, цикл заканчивается при достижении обычного **end do**.

SCHEDULE - конкретный способ распределения итераций. В качестве опций **schedule** используются следующие команды:

STATIC[,m] - блочно-циклическое рапределение итераций, первый блок из m итераций выполняет первая нить, второй блок из m итераций - вторая нить и так далее.

DYNAMIC[,m] - динамическое распределение итераций. Сначала все нити получают порции из m итераций, когда очередная нить заканчивает свою работу – она получает следующую порцию.

GUIDED[,m] - при необходимости, размер блока уменьшается начиная с величины 1 до m.

RUNTIME - размер блока выбирается во время выполнения. По умолчанию, в конце цикла происходит неявная синхронизация. Синхронизацию можно отменить, используя команду **ENDDO NOWAIT**.

Для бригады из p нитей и цикла из n итераций пусть n/p будет целым числом q, которое удовлетворяет n = p*q - r, и 0 <= r < p. Если в опции **STATIC,m** значение m не задано, значение m будет равно q.

В следующем примере, оператор Print 1 напечатает либо 2, либо 5, в зависимости от времени выполнения каждой нити и от реализации OpenMP. По двум причинам X может не равняться 5. Во-первых, Print 1 может выполнится перед присвоением 5. Во-вторых, даже если Print 1 выполнится после операции присвоения, нет гарантии, того что мы увидим его, так как нить 1 не обязательно будет выполняться первой (смотри пример P2). Барьер после Print 1 вызывает точку синхронизации, таким образом гарантировано, что Print 2 и Print 3 напечатают 5.

Пример 10.5

```
   program p5
     use omp_lib
     integer x
     x = 2
  !$omp parallel num_threads(2) shared(x)
    if (omp_get_thread_num() .eq. 0) then
      x = 5
    else
! print чтение x зависит от того, какая нить выполнится раньше
     print *,"1: thread#", omp_get_thread_num(), "x = ", x
    endif

  !$omp barrier

    if (omp_get_thread_num() .eq. 0) then
! print  2
     print *,"2: thread# ", omp_get_thread_num(), "x = ", x
    else
! print 3
      print *,"3: thread# ", omp_get_thread_num(), "x = ", x
    endif

  !$omp end parallel

  end program p5
```

Результат работы программы приведен ниже:

```
 1: THREAD#            1 X =            5
 2: THREAD#            0 X =            5
 3: THREAD#            1 X =            5
```

Если в программе есть только один параллельный цикл и нет других директив, которые должны выполняться параллельно, можно воспользоваться конструкцией

```
 !$omp parallel do [опция[[,] опция] ...]
      цикл-do
 [!$omp end parallel do]
```

Опции в директиве **!$omp parallel do** такие же, как и в **!$omp do**.

Пример 10.6 Параллельный цикл

```
!$omp parallel do schedule (static,2)
   do i = 1,n
     do j = 1,m
       A(i,j)=B(i,j-1)+B(i-1,j))/2.0
     enddo
   enddo
!$omp end parallel do
```

В Примере 10.6 внешний цикл является параллельным, а внутренний последовательным.

Следующий пример демонстрирует работу параллельного цикла. Начальная нить выполняется последовательно до тех пор, пока она не достигнет директивы **PARALLEL**. В этой точке начальная нить создаст бригаду из 20 нитей. Эта бригада будет состоять из мастер нити, имеющей 0 номер и 19 рабочих нитей. Все задания в бригаде будут выполняться параллельно. 1000 итераций цикла будет поделена между 20 нитями, таким образом, каждая нить получит по 50 итераций. Когда все нити закончат свою работу будет выполнена синхронизация в конце цикла.

Пример 10.7

```
   program P6
      use omp_lib
      parameter (n=1000)
      integer n, i
      real a(n), b(n), c(n)

       call omp_set_dynamic (.false.)
       call omp_set_num_threads (20)
 ! initialize arrays a and b.
      do i = 1, n
       a(i) = i * 1.0
       b(i) = i * 2.0
     enddo

 ! compute values of array c in parallel.
!$omp parallel shared(a, b, c), private(i) !$omp do
     do i = 1, n
       c(i) = a(i) + b(i)
     enddo
!$omp end parallel
```

```
    write(*,*) ' c(20) = ', c(20)
  end program p6

  c(20) = 60
```

Создание секции. Секция создается для выполнения нескольких структурированных блоков неитеративного типа, которые могут выполняться одновременно разными нитями. Каждый структурированный блок выполняется одной нитью. Синтаксис этого оператора следующий:

```
!$omp sections [опция[[,] опция] ...]
  [!$omp section]
    структурированный-блок
   [!$omp section structured-block ]
   ...
   !$omp end sections [nowait]
```

опциями могут быть:

- private(список)
- firstprivate(список)
- lastprivate(список)
- reduction(оператор|имя процедуры:список)

Секции выполняются только внутри параллельной области. И только нити параллельной бригады участвуют в выполнении структурированных блоков. В конце области существует неявный барьер.

Параллельная секция создается с помощью следующей директивы:

Пример 10.8

```
!$omp parallel sections private(u,z), shared(x,y) num_threads(2)
  call work(a,b,ix_thread,x)
  call work(a,b,ix_thread,y)
!$omp end parallel
```

Создание параллельной секции **!$omp parallel section**, как и в случае параллельного цикла осуществляется в том случае, когда нет других директив, которые должны выполняться параллельно. Все секции будут выполняться параллельно - каждая своей нитью. В качестве опций можно использовать **private, shared, if** и некоторые другие опции. (Операторы, заключенные в квадратные скобки могут быть опущены).

Пример 10.9

```
 subroutine p8()
!$omp parallel sections
!$omp section
   call xaxis()
!$omp section
   call yaxis()
!$omp section
   call zaxis()
!$omp end parallel sections
end subroutine p8
```

В примере 10.9 в параллельной области создаются три секции, в каждой из которых выполняется своя подпрограмма.

В параллельной секции можно использовать условный оператор

```
IF(скалярное_логическое_выражение)
```

Если используется условный оператор **IF**, то соответствующая параллельная область программы будет выполняться несколькими нитями только при условии, что скалярное логическое выражение имеет значение истина. Например, указав IF(N $>$ 10000), можно заставить компилятор породить такие коды, что во время выполнения они проверяли бы размерность (N) и выполнялись бы параллельно, только если эта размерность достаточно велика - больше 10000. Это позволяет избегать распараллеливания при маленьких размерностях, когда оно может оказаться невыгодным из-за большой доли накладных расходов.

При программировании в OpenMP должны выполняться следующие правила [27]:

1. Параллельные секции могут быть вложены друг в друга.

2. Число нитей в параллельной секции, в зависимости от условий можно менять.

3. Параллельное исполнение нитей можно менять динамически.

   ```
   !$omp parallel if(условие)
   ```

   Если (условие) ложно - параллельные секции выполняться не будут.

В примере 10.10 в параллельной области создается 4 нити. В каждой параллельной секции нить обращается к подпрограмме CUBE, которая возводит переданное ей целое число в куб.

Пример 10.10

```
    program P9
       use omp_lib
       integer square
       integer x, y, z, w, xs, ys, zs, ws
       call omp_set_dynamic (.false.)
       call omp_set_num_threads (4)

       x = 2
       y = 3
       z = 5
       w = 7
       open(7,file='omp.txt')
!$omp parallel
!$omp sections
!$omp section
       xs = cube(x)
       write(7,*) "id = ", omp_get_thread_num(), "xs =", xs
  !$omp section
      ys = cube(y)
      write(7,*) "id = ", omp_get_thread_num(), "ys =", ys
  !$omp section
      zs = cube(z)
      write(7,*) "id = ", omp_get_thread_num(), "zs =", zs
  !$omp section
      ws = cube(w)
      write(7,*) "id = ", omp_get_thread_num(), "ws =", ws
  !$omp end sections
!$omp end parallel
     end program p9

     integer function cube(n)
        integer n
        cube = n*n*n
     end
```

Результаты работы программы P9.

```
id =            0 xs =           8
id =            3 ws =         343
id =            2 zs =         125
id =            1 ys =          27
```

В программу на OpenMP можно вкладывать часть кода, написанного на языке ассемблер. Приведем пример программирования на ассемблере:

Пример 10.11

```
if( omp_get_thread_num().EQ.5 ) then
  код на ассемблере для нити 5
else
  код для остальных нитей
endif
```

Для создания последовательного участка программы, который будет исполняться основной нить программы, используется следующие директивы:

```
!$omp single [опция][,опция]
    блок операторов, выполняющихся последовательно
!$omp end single
```

Эти директивы являются неявной точкой синхронизации. Все параллельные нити завершаются до выполнения последовательного участка. В качестве опций могут быть использованы следующие команды:

- private(список)
- firstprivate(список)

Для создания точек синхронизации в OpenMP существует несколько директив. Директива **!$omp barrier** позволяет установить барьер и организовать синхронизацию потоков. Директива **!$omp atomic** позволяет обеспечить "атомистическую"синхронизацию, предохраняющую от одновременной записи несколькими нитями в одно и то же поле оперативной памяти. Это позволяет предотвратить ситуацию гонки. Например, если запись производится в элемент массива a(index(i)), где i-управляющая переменная (счетчик) распараллеленного цикла, то нельзя заранее - на этапе трансляции - предсказать, не будет ли на этапе выполнения попытки одновременной записи разными нитями, имеющими

разные i, в один и тот же элемент массива a. Чтобы избежать этого, следует сделать так, как это указано в следующем примере:

```
!$omp atomic
      a(index(i)) = a(index(i)) + b(i)
      !$omp flush
```

Без этой синхронизации было бы непонятно, с каким значением a(index(i)) произведено сложение. Не менее важной является директива **!$omp flush**. Эта директива позволяет синхронизировать кэши процессоров и согласовать состояние оперативной памяти (при этом переменные из регистров будут записаны в память, сбросятся буферы записи и т.д.).

Пример 10.12

```
program P10
  real x(1000), y(10000)
  integer index(10000)
  integer i
 do i=1,10000
   index(i) = mod(i, 1000) + 1
   y(i) = 0.0
 enddo
 do i = 1,1000
   x(i) = 0.0
 enddo
 call work3(x, y, index, 10000)
end program P10

real function work1(i)
  integer i
  work1 = 1.0 * i
  return
end function work1

real function work2(i)
  integer i
  work2 = 2.0 * i
  return
end function work2
```

```
subroutine work3(x, y, index, n)
  real x(*), y(*)
  integer index(*), n
  integer i
!$omp parallel do shared(x, y, index, n)
  do i=1,n
!$omp atomic
    x(index(i)) = x(index(i)) + work1(i)
    y(i) = y(i) + work2(i)
  enddo
end subroutine work3
```

В примере 10.10 показано как избежать условий гонки процессоров (т.е. одновременной модернизации элемента x параллельными нитями), с помощью конструкции **!$omp atomic**. Преимущество в использовании этой конструкции заключается в том, что она позволяет обновить два различных элемента x при параллельном выполнении.

Директива **!$omp flush[(список)]**.

OpenMP является системой с общей памятью, которая распределяется между всеми параллельными нитями. Каждая нить имеет свой собственный локальный участок памяти и все локальные переменный записываются в эту локальную память и недоступны другим нитям. Поэтому, для синхронизации параллельных вычислений, необходимо записывать в оперативную память все сделаные изменения.

Создать синхронизированные переменные в разных нитях можно с помощью следующих условий:

- Создать глобально видимые переменные, общие блоки и модули.
- Локальные переменные, которые не имеют атрибут **SAVE**, но адрес которых мы знаем.
- Местные переменные, которые не имеют атрибут **SAVE**, но которые объявлены с атрибутом **SHARED** в параллельной области.
- Параметры Dummy.
- Все разыменованные указатели.

Необязательный параметр **список** должен содержать имена переменных, разделенных запятыми, которые должны быть синхронизированы. Неявно, директива **!$omp flush**, создается директивами

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED

В примере 10.11 показана синхронизация особых объектов между нитями:

Пример 10.13

```
  program P12
    use omp_lib
    integer isync(256)
    real work(256)
    real result(256)
    integer iam, neighbor
  !$omp parallel private(iam, neighbor) shared(work, isync)
    iam = omp_get_thread_num() + 1
    isync(iam) = 0
  !$omp barrier

! Выполнение своей части работы с массивом

    work(iam) = fn1(iam)

! Завершение своей части работы

!  Первый  flush гарантирует, что работа сделана и занесена в
!массив до  синхронизации.  Второй flush гарантирует, что
!синхронизация  прошла.

  !$omp flush(work,isync)
    isync(iam) = 1
```

```
!$omp flush(isync)

  if (iam .eq. 1) then
    neighbor = omp_get_num_threads()
  else
    neighbor = iam - 1
  endif
  do while (isync(neighbor) .eq. 0)
!$omp flush(isync)
  end do
!$omp flush(work, isync)
  result(iam) = fn2(work(neighbor), work(iam))
!$omp end parallel
end program P12

real function fn1(i)
  integer i
  fn1 = i * 2.0
  return
end function fn1

real function fn2(a, b)
  real a, b
  fn2 = a + b
  return
end function fn2
```

Директива **threadprivate** задает память для копий глобальных объектов, которая имеется во всех нитях. Каждая нить имеет доступ к этому типу памяти, и к ней нельзя обратиться из других нитей. Такая память называется **threadprivate** памятью. Эта директива должна появляться в разделе объявлений процедуры, после объявления общих областей. Каждая нить получает свою собственную копию общих областей, таким образом, данные, записанные в общую область одной нитью не видны другими нитями. В последовательной области и MASTER разделах программы, доступ существует к копии главной нити общей области.

Синтаксис директивы **threadprivate**

```
!$omp threadprivate(список).
```

Здесь список - это список переменных и массивов, разделенных запятой. Каждая копия объекта **threadprivate** инициализируется один раз, но в не указанном пункте программы, а при первой ссылке.

Пример 10.14

```
 program P13
      use omp_lib
      integer, allocatable, save :: a(:)
      integer, pointer, save :: ptr
      integer, save :: i
      integer, target :: targ
      logical :: firstin = .true.
!$omp threadprivate(a, i, ptr)
      allocate (a(4))
      a = (/1,2,3,5/)
      ptr => targ
      i = 5
!$omp parallel copyin(i, ptr)
!$omp critical
     if (firstin) then
        targ = 7    ! update target of ptr
        i = i + 10
      if (allocated(a)) a = a + 10
        firstin = .false.
      end if
      if (allocated(a)) then
        print *, 'a = ', a
      else
        print *, 'a is not allocated'
      end if
       print *, 'ptr = ', ptr
       print *, 'i = ', i
       print *
!$omp end critical
!$omp end parallel
       read(*,*)
    end program P13
```

При выполнении этой программы на двух процессорах получится один из следующих вариантов: либо

```
 a = 11 12 13 15
 ptr = 7
 i = 15

 A is not allocated
  ptr = 7
  i = 5

 или

 A is not allocated
  ptr = 7
  i = 15

  a = 1 2 3 5
  ptr = 7
  i = 5
```

На четырехядерном процессоре мы получим следующие варианты (приведены только два варианта):

```
 a =           11          12          13          15
 ptr =            7
 i =           15

 A is not allocated
 ptr =            7
 i =            5

 A is not allocated
 ptr =            7
 i =            5

 A is not allocated
 ptr =            7
 i =            5

A is not allocated
 ptr =            7
 i =           15
```

```
 a =            1           2           3           5
 ptr =            7
 i =            5

 A is not allocated
 ptr =            7
 i =            5

 A is not allocated
 ptr =            7
 i =            5
```

Директива **!$omp REDUCTION(operator|intrinsic:список)**. Эта директива позволяет сократить те переменные, которые появляются в списке **список** опертора **operator**, где **operator** это +, *, -, .AND., .OR., .EQV., or .NEQV., или MAX, MIN, IAND, IOR, or IEOR.

Переменные, которые появляются в пункте REDUCTION, должны быть с атрибутом SHARED в программе. Копия каждой переменной в списке создается для каждой нити так, как будто используется атрибут PRIVATE.

Для тех, кто программирует на C++ приведем пример программы с использованием OpenMP. Директива OpenMP начинается с атрибута pragma. В примере 10.15 приведена программа с директивой **atomic**, аналог программы 10.12

Пример 10.15

```
#include <omp.h>
 float work1(int i)
 {
   return 1.0 * i;
 }

 float work2(int i)
  {
    return 2.0 * i;
  }

  void a16(float *x, float *y, int *index, int n)
  {
```

```
   int i;
#pragma omp parallel for shared(x, y, index, n)
   for (i=0; i<n; i++)
    {
#pragma omp atomic
   x[index[i]] += work1(i);
   y[i] += work2(i);
    }
 }

 int main()
 {
   float x[1000];
   float y[10000];
   int index[10000];
   int i;

   for (i = 0; i < 10000; i++) {
     index[i] = i % 1000;
     y[i]=0.0;
 }
  for (i = 0; i < 1000; i++)
    x[i] = 0.0;

    a16(x, y, index, 10000);
    return 0;
}
```

10.4.3. MPI

MPI (Message Passing Interface - интерфейс передачи сообщений) является сегодня самой распространенной системой параллельного программирования. С помощью MPI можно создавать надежные и эффективные параллельные программы для многопроцессорных ЭВМ. С помощью MPI в программе можно организовать взаимодействие параллельных процессов. Существует несколько версий стандарта MPI – версия 1.0, 1.1. В 1998 году появился стандарт MPI 2.0. Этот стандарт значительно развил и усилил возможности пакета.

На сегодняшний день MPI поддерживается двумя языками – Фор-

траном и C++. MPI представляет собой библиотеку функций, которая содержит более 150 процедур. Технологию MPI используют для программирования на системах с локальной памятью. MPI реализован для всех операционных систем, процессоров и компиляторов. В отличии от системы параллельного программировании OpenMP, в MPI используется система передачи сообщений между отдельными ветвями программы, которые исполняются на разных процессорах, имеющих свою локальную память.

В основу MPI были положены четыре основные концепции:

- Тип данных, пересылаемых в сообщении.
- Тип операции передачи сообщения.
- Понятие коммуникатора (группы процессов).
- Понятие виртуальной топологии.

Директивы MPI имеют префикс $MPI\_$ директива. Кроме того, принято, что в названиях функций начальные буквы пишутся заглавными буквами, а все остальные – прописными. Все константы записываются заглавными буквами. MPI представляет из себя библиотеку подпрограмм и функций, поэтому необходимо подключить соответствующую библиотеку, которая находятся в файле mpif.h. Таким образом, в MPI программу необходимо включить файл:

```
include 'mpif.h'
```

Каждому процессу в MPI выделяется свое адресное пространство. Глобальных переменных не существует, поэтому весь обмен информацией осуществляется с помощью явной передачи сообщений. Процессы в MPI принадлежат разным группам. Внутри группы они нумеруются целыми числами, начиная с нуля. В начальной группе содержаться все процессы MPI. При передаче сообщений используется адресация по номеру процесса.

В библиотеку MPI входит:

- процедуры, которые начинают и завершают MPI процессы;
- процедуры, которые реализуют обмен сообщениями типи точка-точка;
- процедуры, которые осуществляют коллективные операции;

- процедуры, которые предназначены для работы со структурами данных;

- процедуры, предназначенные для задания топологии вычислительных процессов.

Рассмотрим основные процедуры и команды пересылки сообщений в MPI. Программа, использующая MPI, обязана начинаться с вызова инициализирующей функции MPI, то есть с вызова функции MPI_Init. При этом будет создана группа процессов, в которой будут находится все процессы создаваемой программы и область связи, которая описывается коммуникатором MPI_COMM_WORLD. При старте MPI программы создается коммуникатор с именем MPI_COMM_WORLD, который существует во все время жизни программы и необходим для взаимодействия всех запущенных процессов. Целочисленный параметр IERROR служит для возврата кода ошибки, если она возникает при старте MPI.

```
INTEGER IERROR
CALL MPI_INIT(IERROR)
```

Для запуска откомпилированной программы на N процессорах необходимо выполнить команду:

```
mpirun -np N <имя программы>
```

Если мы запускаем программу на N процессорах, то номера процессов изменяются в пределах от 0 до N-1.

Основой MPI является посылка сообщений от одного процесса к другому. Рассмотрим подробнее процесс передачи сообщения от одного процесса другому или целой группе процессов. **Сообщение - это совокупность данных некоторого вида.** Сообщение должно иметь несколько атрибутов: номер процеса - получателя, номер процесса отправителя, идентификатор сообщения и ряд другой информации. В конце программы с использованием MPI должна быть завершающая команда:

```
 CALL MPI_FINALIZE(IERROR)
```

К этому моменту все сообщения между процессами и все процессы должны быть завершены. Приведем пример простой программы на Фортране с использованием MPI [29]:

Пример 10.16

```
program MPI1
  include 'mpif.h'
  integer ierr
  print *, 'Before MPI_INIT'
  call MPI_INIT(ierr)
  print *, 'Parallel section'
  call MPI_FINALIZE(ierr)
  print *, 'After MPI_FINALIZE'
end program MPI1
```

В данной программе, в зависимости от реализации MPI, текст $BeforeMPI\_INIT$ и $AfterMPI\_FINALIZE$ будут выводить либо все процессы программы, либо один выделенный процесс. Строчку *Parallel section* должны вывести все запущенные процессы. Порядок вывода этой информации от разных процессов будет в общем случае произвольным.

Для проверки того, работаем ли мы в параллельной секции или в последовательной, используется команда:

```
INTEGER IERR
LOGICAL FLAG
MPI_INITIALIZED(FLAG, IERR)
```

Данная команда возвращает .TRUE. если она была вызвана из параллельной части программы и .FALSE. – если была вызвана из последовательной части.

Определение общего числа параллельных процессов в группе **COMM** осуществляется с помощью команды:

```
INTEGER COMM, SIZE, IERROR
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
```

где **COMM** - идентификатор группы, **SIZE** - размер группы.

Определение номера процесса в группе:

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
```

где **COMM** - идентификатор группы, **RANK** – номер процесса, вызвавшего процедуру. Процедура возвращает номер процесса, вызвавшего ее.

В следующем примере приведена программа, которая выдает на экран номер процесса и число процессов в данном коммуникаторе.

Пример 10.17

```
program MPI2
  include ‘mpif.h’
  integer ierr, size, rank
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  print *, ‘process ’, rank, ‘, size ’, size
  call MPI_FINALIZE(ierr)
end program MPI2
```

Прием и передача сообщений осуществляется с помощью соответствующих процедур. Передача сообщений

```
      INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
      type BUF()
      CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

- **BUF** – адрес начала расположения пересылаемых данных;
- **COUNT** – число пересылаемых элементов;
- **DATATYPE** – тип посылаемых элементов;
- **DEST** – номер процесса-получателя в группе, связанной с коммуникатором comm;
- **TAG** – идентификатор сообщения;
- **COMM** – коммуникатор области связи.

Прием сообщений осуществляется с помощью процедуры:

```
      type BUF
      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
      MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS, IERROR)
```

- **BUF** – адрес начала расположения принимаемого сообщения;
- **COUNT** – максимальное число принимаемых элементов;
- **DATATYPE** – тип элементов принимаемого сообщения;

- **SOURCE** – номер процесса-отправителя;
- **TAG** – идентификатор сообщения;
- **COMM** – коммуникатор области связи;
- **STATUS** – атрибуты принятого сообщения.

Приведенные выше основные шесть процедур (за исключением процедуры $MPI\_INITIALIZED(FLAG, IERR)$ позволяют осуществить распараллеливание простых программ [30]. Приведем еще одну полезную функция, которая позволяет определить время, затраченное на выполнение программы.

```
real(8) MPI_WTIME(IERROR), time_start
time_start = MPI_WTIME(IERROR)
```

Данная функция возвращает время в секундах. Если вызвать такую функцию дважды – до начала работы программы и перед ее завершением, то разность второго и первого значений даст нам время в секундах, затраченное на выполнение программы.

Ниже приведен текст программы, которая определяет имена процессоров, вызывающие процессы и время выполнения.

Пример 10.18

```
program MPI3
  include 'mpif.h'
  integer ierr, rank, len, i, NTIMES
  parameter (NTIMES = 100)
  character*(MPI_MAX_PROCESSOR_NAME) name
  double precision time_start, time_finish, tick
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_GET_PROCESSOR_NAME(name, len, ierr)
  tick = MPI_WTICK(ierr)
  time_start = MPI_WTIME(ierr)
 do i = 1, NTIMES
    time_finish = MPI_WTIME(ierr)
 end do
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    print *, 'processor ', name(1:len),
```

```
      &       ', process ', rank, ': tick = ', tick,
      &       ', time = ', (time_finish-time_start)/NTIMES
     call MPI_FINALIZE(ierr)
   end program MPI3
```

В приведенном ниже примере происходит обмен между процессом с четным номером и процессором на единицу большим [29]. Для процесса с максимальным номером осуществляется проверка для того, что бы не было посылки информации несуществующему процессу. Переменная b изменяется на процессорах, имеющих нечетный номер.

Пример 10.19

```
 program MPI4
   include 'mpif.h'
   integer ierr, size, rank, a, b
   integer status(MPI_STATUS_SIZE)
   call MPI_INIT(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   a = rank
   b = -1
 if(mod(rank, 2) .eq. 0) then
  if(rank+1 .lt. size) then
   call MPI_Send(a, 1, MPI_INTEGER, rank+1,5, &
                 MPI_COMM_WORLD,ierr)
  end if
 else
   call MPI_Recv(b, 1, MPI_INTEGER, rank-1, 5, &
   MPI_COMM_WORLD, status, ierr)
 end if
   print *, 'process ', rank,'a = ', a, ', b = ', b
   call MPI_FINALIZE(ierr)
end
```

В следующем примере приводится программа, которая позволяет экспериментально определить время, необходимое на пересылку информации в зависимости от длины посылаемого сообщения [29]. Таким образом можно оценить основные характеристики используемой сети: время, затраченное на передачу сообщения нулевой длины (латентность сети), максимальную пропускную способность сети и длину сообщения на котором она достигается. С помощью константы **NMAX** можно задать

ограничения на максимальную длину сообщения. Другая константа – **NTIMES** ограничивает количество повторений таких сообщений. В программе P18 посылается сообщение нулевой длины, затем посылается сообщение длиной 8 байт, а затем количество байт информации удваивается до **NMAX**.

Пример 10.20

```
 program MPI5
   include 'mpif.h'
   integer ierr, rank, size, i, n, lmax, NMAX, NTIMES
   parameter (NMAX = 1 000 000, NTIMES = 10)
   double precision time_start, time, bandwidth, max
   real*8 a(NMAX)
   integer status(MPI_STATUS_SIZE)
   call MPI_INIT(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   time_start = MPI_WTIME(ierr)
   n = 0
   max = 0.0
   lmax = 0
  do while(n .le. NMAX)
     time_start = MPI_WTIME(ierr)
    do i = 1, NTIMES
     if(rank .eq. 0) then
       call MPI_SEND(a, n, MPI_DOUBLE_PRECISION, 1, 1, &
                     MPI_COMM_WORLD, ierr)
       call MPI_RECV(a, n, MPI_DOUBLE_PRECISION, 1, 1, &
       MPI_COMM_WORLD, status, ierr)
     end if
     if(rank .eq. 1) then
       call MPI_RECV(a, n, MPI_DOUBLE_PRECISION, 0, 1, &
       MPI_COMM_WORLD, status, ierr)
       call MPI_SEND(a, n, MPI_DOUBLE_PRECISION, 0, 1, &
       MPI_COMM_WORLD, ierr)
     end if
    enddo
       time = (MPI_WTIME(ierr)-time_start)/2/NTIMES
       bandwidth = (8*n*1.d0/(2**20))/time
     if(max .lt. bandwidth) then
```

```
        max = bandwidth
        lmax = 8*n
       end if
       if(rank .eq. 0) then
        if(n .eq. 0) then
          print *, 'latency = ', time, ' seconds'
        else print *, 8*n,' bytes, bandwidth =', bandwidth, &
          ' Mb/s'
        end if
       end if
       if(n .eq. 0) then
         n = 1
       else
         n = 2*n
       end if
      end do
      if(rank .eq. 0) then
         print *, 'max bandwidth =', max, &
                  ' Mb/s , length =', lmax, ' bytes'
      end if
        call MPI_FINALIZE(ierr)
    end program MPI5
```

Более подробно с технологий программирования MPI можно познакомится в работах [26], [28], [29], [30], [34], [35] и других.

Глава 11

Введение в параллельные численные методы

11.1. Введение

Главной движущей силой развития многопроцессорных ЭВМ является потребность в научных вычислениях, методах математического моделирования и т.д. Одновременно происходит значительное улучшение математического аппарата, призванного решать поставленные задачи. Создание мощных многопроцессорных компьютеров вызвало необходимость усовершенствовать и численные методы решения задач, построить параллельные алгоритмы существующих методов решения. Теория построения параллельных алгоритмов зародилась достаточно давно, в конце позапрошлого столетия [32], реальные многопроцессорные ЭВМ появились только в 70 годы прошлого столетия. Одними из первых были 64 - процессорный компьютер ILLIAC IV и векторные компьютеры фирм Texas Instruments, Control Data Corporation, Cray Research Corporation.

В последнее время развитие многопроцессорных ЭВМ происходит как по линии наращивания производительности отдельных процессоров, из не очень большого числа которых состоят компьютеры типа Cray, так и увеличением числа относительно простых процессоров в одной ЭВМ.

11.2. Теоретические основы параллельных методов

Для построения параллельных программ необходимо провести анализ используемых алгоритмов с целью возможности оценки степени их распараллеливания. Для этого используется модель в виде графа [33] и [6]. Будем предполагать, для простоты, что время выполнения любой операции в программе одинаково и равно единице (в какой-либо системе единиц измерения). Примем, что время передачи сообщений между процессорами пренебрежимо мало. Тогда все операции, которые будут исполняться в данном алгоритме, и информационные зависимости, существующие между операциями можно представить в виде ориентированного графа.

$$G = (V, R). \tag{11.1}$$

Здесь $V = \{1, \ldots, |V|\}$ – множество вершин графа. R - множество дуг графа, при этом полагается, что если операция j использует результат операции i, тогда и дуга r=(i,j) принадлежит графу. Обозначим диаметр графа, то есть длину максимального пути через d(G).

Тогда, если между некоторыми операциями алгоритма нет дуги, то они могут быть выполнены параллельно. Приведем способ описания параллельного выполнения алгоритм, предложенный в [33] и [6]. Пусть есть p свободных процессоров, предназначенных для выполнения данного алгоритма. Для параллельной реализации вычислений необходимо задать некоторое множество (называемое расписанием):

$$H_p = \{(i, P_i, t_i) : i \in V\}. \tag{11.2}$$

В (11.2) для каждой операции $i \in V$ указан номер используемого процессора P_i и время начала выполнения операции. t_i. Необходимо выполнить ряд требований над множеством H_p для того, чтобы расписание было реализуемым.

- $\forall i, j \in V : t_i = t_j \Rightarrow P_i \neq P_j$, один процессор не может быть задействован в один и тот же момент времени для разных операций,
- $\forall (i, j) \in R \Rightarrow t_j \geq t_i + 1$, все необходимые данные должны быть вычислены к моменту начала операций.

Таким образом, схема алгоритма G, вместе с расписанием H_p для p процессоров рассматривается как модель параллельного алгоритма $A_p(G, H_p)$. Максимальное значение времени в расписании определяет время выполнения параллельного алгоритма:

$$T_p(G, H_p) = \max_{i \in V}(t_i + 1). \tag{11.3}$$

Найти минимальное время выполнения алгоритма можно, если использовать расписание с минимальным временем выполнения:

$$T_p(G) = \min_{H_p} T_p(G, H_p). \tag{11.4}$$

Если выбрать наиболее быстродействующий алгоритм, то можно также сократить время его выполнения:

$$T_p = \min_{G} T_p(G). \tag{11.5}$$

Значения $T_P(G, H_p)$, $T_p(G)$, T_p из (11.3-11.5) можно использовать для оценки выбора налучшего по быстродействию параллельного алгоритма. Введем понятие наиболее быстро выполняемого алгоритма, обозначив $T_\infty = \min_{p \geqslant 1} T_p$.

Таким образом T_∞ представляет нижнюю грань возможного времени выполнения параллельного алгоритма, если в нашем распоряжении находится вычислительная система с бесконечным количеством процессоров.

Рассмотрим, без доказательств, некоторые теоретические оценки, приведенные в [33].

Теорема 11.1 *Минимально возможное время выполнения параллельного алгоритма определяется длиной максимального пути вычислительной схемы алгоритма, т.е.*

$$T_\infty(G) = d(G). \tag{11.6}$$

Теорема 11.2 *Пусть для некоторой вершины вывода в вычислительной схеме алгоритма существует путь из каждой вершины ввода. Кроме того, пусть входная степень вершин схемы (количество входящих дуг) не превышает 2. Тогда минимально возможное время выполнения параллельного алгоритма ограничено снизу значением*

$$T_\infty(G) = \log_2 \ n. \tag{11.7}$$

где n есть количество вершин ввода в схеме алгоритма.

Теорема 11.3 *При уменьшении числа используемых процессоров время выполнения алгоритма увеличивается пропорционально величине уменьшения количества процессоров, т.е.*

$$\forall q = cp,\ 0 < c < 1 \Rightarrow\ T_p \leqslant cT_q. \tag{11.8}$$

Теорема 11.4 *Для любого количества используемых процессоров справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма*

$$\forall p \Rightarrow T_p < T_\infty + T_1/p. \tag{11.9}$$

Теорема 11.5 *Времени выполнения алгоритма, которое сопоставимо с минимально возможным временем T_∞, можно достичь при количестве процессоров порядка $p \sim T_1/T_\infty$,* а именно:

$$p \geqslant T_1/T_\infty \Rightarrow T_p \leqslant 2T_\infty. \tag{11.10}$$

Из приведенных теорем можно сделать следующие выводы:

1. Из **Теоремы 11.1** – при выборе параллельного алгоритма нужно выбирать граф с минимально возможным диаметром.
2. Из **Теоремы 11.5** – оптимальное количество процессоров можно найти как $p \sim T_1/T_\infty$.
3. Из **Теорем 11.4 и 11.5** – время выполнения алгоритма ограничено сверху (11.4)-(11.5).

Рассмотрим проблемы, которые возникают при создании параллельных методов [34]. Для этого возьмем простую задачу нахождения общей суммы некоторого набора чисел:

$$S = \sum_{i=1}^{n} x_i.$$

Последовательный алгоритм состоит в почленном суммировании элементов множества x_i. Схематически этот алгоритм может быть представлен в следующем виде:

$$G_1 = (V_1, R_1). \tag{11.11}$$

В уравнении (11.11) значения $V_1 = \{v_{01}, \ldots, v_{0n}, v_{11}, \ldots, v_{1n}\}$ - представляют множество операций, причем вершины $\{v_{01}, \ldots, v_{0n}\}$ обозначают операции ввода. Вершина $v_{1i},\ i \leqslant i \leqslant n$ означает прибавление очередного значения x_i к сумме S. Информационные зависимости, определяющие операции над числами в графе (11.11) задаются множеством дуг

$$R_1 = \{(v_{0i}, v_{1i}), (v_{1i}, v_{1i+1}),\quad 1 \leqslant i \leqslant n-1\} \tag{11.12}$$

Приведенный алгоритм не поддается распараллеливанию, так как задает строго параллельное выполнение операций. Распараллелить вычисления можно, если изменить схему вычислений. Новый алгоритм, известный как *каскадный* можно представить в виде:

- сначала все числа рабиваются на пары чисел и для каждой пары выполняется суммирование.
- все полученные на первом этапе числа опять разбиваются на пары, снова выполняется суммирование и т.д.

Вычислительный граф, для $n = 2^k$, может быть представлен в виде:

$$G_2 = (V_2, R_2), \tag{11.13}$$

где $V_2 = \{(v_{1i}, \ldots, v_{il_i}),\ 0 \leqslant i \leqslant k,\ 1 \leqslant l_i \leqslant 2^{-i}n\}$ - вершины графа $\{v_{01}, \ldots, v_{0n}\}, \{v_{l1}, \ldots, v_{ln/2}\}$ операции первой итерации. Соотношение (11.14)

$$R_2 = \{(v_{i-1,2j-1}, v_{i,j}), (v_{i-1,2j}, v_{i,j}),\ 1 \leqslant i \leqslant k,\ 1 \leqslant j \leqslant 2^{-i}n\}. \tag{11.14}$$

задает множество дуг графа.

Количество итераций каскадной схемы равно $k = \log_2 n$. Общее количество операций суммирования равно:

$$K_{seq} = n/2 + n/4 + \ldots + 1 = n - 1.$$

Если выполнить суммирование на параллельном компьютере с достаточным количеством процессоров, количество итераций каскадной схемы станет равным:

$$K_{par} \quad = \quad \log_2 n.$$

Мы приняли, что время выполнения каждой операции одинаково и равно единице, отсюда следует, что ускорение параллельной схемы составляет:

$$S_p = \frac{T_1}{T_p} = \frac{(n-1)}{\log_2 n},$$

а эффективность:

$$E_p = \frac{T_1}{pT_p} = \frac{(n-1)}{p\log_2 n} = \frac{(n-1)}{((n/2)\log_2 n)}.$$

Из полученных выше оценок следует, что время параллельного выполнения совпадает с оценкой, полученной в **Теореме 11.2**.

11.3. Параллельные алгоритмы векторно - матричных операций

Векторные и матричные операцмм являются основными операциями научных и инженерных расчетов, численных методов. Матричные операции занимают много процессорного времени, поэтому распараллеливание таких задач позволит существенно сократить общеее время расчета. Существует много стандартых библиотек, в которых находятся готовые к использованию программы векторно – матричных операций, в том числе и параллельных. Однако такие задачи позволяют продемонстрировать приемы и методы параллельного программирования.

Рассмотрим алгоритм умножения матрицы на вектор $c = A \cdot b$. Здесь c результирующий вектор, A исходная матрица, b - вектор сомножитель.

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \ 1 \leqslant i \leqslant m. \tag{11.15}$$

Разобъем исходную задачу на процессы. Один из процессов будет мастер процессом (master), который распределяет работу между другими подчиненными (slave) процессами и управляет ими. Задача умножения матрицы на вектор состоит из m однотипных операций умножения строки матрицы A на вектор b. Общее количество операций равно $T_1 = m \cdot (2n-1)$.

Последовательный алгоритм умножения матрицы на вектор приведен ниже:

```
do i = 1,m
   c(i) = 0.0
 do j = 1,n
    c(i) = c(i)+a(i,j)*b(j)
 enddo
enddo
```

Временные затраты на этот алгоритм составляют порядка $O(m)$ операций.

Параллельный алгоритм умножения матрицы на вектор будем основывать на разбиении исходной матрицы на строки. Таким образом, у нас будет некоторое количество операций скалярного умножения вектора (строки матрицы) на другой вектор. Если число процессоров меньше числа подзадач, то необходимо объединить несколько подзадач таким образом, чтобы каждый процессор был загружен одинаковым числом таких подзадач.

Оценим время выполнения параллельного алгоритма умножения матрицы на вектор. Выше приведена временная оценка последовательного алгоритма $T_1 = m \cdot n$. Для квадратной матрицы $T_1 = n^2$. Рассмотрим параллельную реализацию матрично - векторного умножения. В этом случае каждый процессор производит умножение своей полосы на вектор b. Размер полос равен n/p, где p - число процессоров. Вычислительная трудоемкость параллельного алгоритма равна таким образом:

$$T_p = \frac{n^2}{p}. \tag{11.16}$$

Ускорение алгоритма в случае его распараллеливания на p процессоров равняется:

$$S_p = \frac{n^2}{n^2/p} = p, \tag{11.17}$$

а эффективность равна

$$E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \tag{11.18}$$

В приведенных оценках не учтено время, затраченное на передачу информации и полагается, что все операции занимают одинаковое время. Тем не менее, (11.17) и (11.18) позволяют приблизительно оценить затраты времени на реализацию параллельного алгоритма.

Рассмотрим программу, реализующую описанный выше параллельный алгоритм умножения матрицы на вектор. В нем главный процесс рассылает подчинненым процессам вектор b. Каждый подчиненный процесс получает строку матрицы A, умножает ее на вектор b и возвращает мастер процессу полученный результат [35].

Программа 11.1 Программа параллельного умножения матрицы на вектор

```
    program parallel_1
      use mpi
      integer MAX_ROWS, MAX_COLS, rows, cols
      integer,parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
! матрица A, вектор b, результирующий вектор c
      real(8) a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_ROWS)
      real(8) buffer (MAX_COLS), ans
      integer myid, master, numprocs, ierr,  &
              status(MPI_STATUS_SIZE)
      integer i, j, numsent, sender, anstype, row
     call MPI_INIT( ierr )
     call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
     call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

! главный процесс - master
       master = 0
 ! количество строк и столбцов матрицы A
       rows = 10
       cols = 100
      if ( myid .eq. master ) then
! master процесс
       do j = 1, cols
          b(j) = j
        do i = 1, rows
           a(i,j) = i
        enddo
       enddo
         numsent = 0
 ! посылка b каждому подчиненному процессу
        call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, &
                       master, MPI_COMM_WORLD, ierr)
 ! посылка строки каждому подчиненному процессу; в TAG -
```

```
номер строки = i
        do i = 1,min(numprocs-1, rows)
         do j = i,cols
           buffer(j) = a(i,j)
         enddo
           call MPI_SEND(buffer, cols, &
           MPI_DOUBLE_PRECISION, i, i, MPI_COMM_WORLD, ierr)
           numsent = numsent + 1
        enddo
! прием результата от подчиненного процесса
        do i = 1, rows
! MPI_ANY_TAG - указывает, что принимается любая строка
           call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, &
           MPI_ANY_SOURCE, MPI_ANY_TAG, &
           MPI_COMM_WORLD, status, ierr)
           sender = status(MPI_SOURCE)
           anstype = status (MPI_TAG)
! определяем номер строки
           c(anstype) = ans
         if (numsent .lt. rows) then
! посылка следующей строки
          do j = 1, cols
              buffer(j) = a(numsent+l, j)
          enddo
            call MPI_SEND (buffer, cols, &
            MPI_DOUBLE_PRECISION, sender, numsent+l, &
            MPI_COMM_WORLD, ierr)
             numsent = numsent+l
          else
! посылка признака конца работы
           call MPI_SEND(MPI_BOTTQM, 0, &
           MPI_DOUBLE_PRECISION, sender, 0, &
           MPI_COMM_WORLD, ierr)
          endif
        enddo
      else
 ! код подчиненного процесса
 ! прием вектора b всеми подчиненными процессами
       call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, &
```

```
      master, MPI_COMM_WORLD, ierr)
! выход, если процессов больше количества строк матрицы
      if (numprocs .gt. rows) goto 200
! прием строки матрицы
90      call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, &
        master, MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
       if (status(MPI_TAG) .eq. 0) then
          go to 200
! конец работы
       else
          row = status(MPI_TAG)
! номер полученной строки
          ans = 0.0
       do i = 1, cols
!скалярное произведение векторов
          ans = ans+buffer(i)*b(i)
       enddo

! передача результата главному процессу
         call MPI_SEND(ans,1,MPI_DOUBLE_PRECISION, &
         master,row, MPI_COMM_WORLD, ierr)
        go to 90
! цикл для приема следующей строки матрицы
        endif
200    continue
      endif
        call MPI_FINALIZE(ierr)
        stop
    end
```

Достаточно просто обобщить алгоритм умножения матрицы на вектор для умножения матрицы на матрицу. Вместо векторов b и c необходимо ввести матрицы B и C. Основной алгоритм остается похожим на алгоритм Программы 11.1. При проведении матричных вычислений приходится производить большое количество однотипных вычислений. В этом случае, как и при умножении матрицы на вектор, существует параллелизм на уровне данных. Распределение вычислений сводится, в основном, к распределению данных по процессорам. В случае умножения матрицы на матрицу данные могут быть распределены либо по отдельным столбцам (или строкам), либо по отдельным блокам. В первом

случае говорят о ленточном распределении данных, во втором о блочном распределении. В обоих случаях исследование информационной зависимости приводит к задачам, в которых обмениваться данными необходимо только между ближайшими соседними блоками.

После определения информационных зависимостей необходимо достаточно равномерно распределить подзадачи между процессорами. При этом возникают некоторые сложности. С одной стороны, для обеспечения равномерной загрузки процессоров, нужно стараться минимизировать обмен информацией между процессорами, с другой стороны, если подзадачи не будут рапределены на все процессоры, трудно ожидать большого быстродействия многопроцессорной системы.

Задача умножения матрицы A размером mxn на матрицу B размером nxk сводится к вычислению матрицы C размером mxk:

$$c_{ij} = \sum_{l=0}^{n-1} a_{il} \cdot b_{lj}, \quad 0 \leqslant i \leqslant m, \; 0 \leqslant j \leqslant k. \tag{11.19}$$

Обычный, последовательный алгоритм умножения матрицы на матрицу может быть записан в виде:

```
do 2 j = 1,n
     do 2 i = 1,n
        s =0.0d0
     do 3 l = 1,n
  3   s = s + a(i,l)*b(l,j)
     c(i,j) = s
  2 continue
```

Более эффективный алгоритм можно представить в виде:

```
 do 22 j = 1,n
     do m = 1,n
       row(m) = a(i,m)
     end do
    do 22 i = 1,n
     c(i,j) =0.0d0
      do 33 l = 1,n
 33     c(i,j) = c(i,j) + row(l)*b(l,j)
 22  continue
```

Программа 11.2 Программа параллельного умножения матрицы на матрицы

```
    program parallel_2
      integer MAX_AROWS, MAX_ACOLS, MAX_BCOLS
      integer,parameter (MAX_AROWS = 20, &
                         MAX_ACOLS = 1000, MAX_BCOLS = 20)
! матрицы A,B,C
      real(8) a(MAX_AROWS,MAX_ACOLS)
      real(8) b(MAX_ACOLS,MAX_BCOLS), c (MAX_AROWS, MAX_BCOLS)
      real(8) buffer (MAX_ACOLS), ans (MAX_ACOLS)
      real(8) starttime, stoptime
      integer myid, master, numprocs, ierr, &
              status(MPI_STATUS_SIZE)
      integer i, j , numsent, sender, anstype, row, &
              arows, acols, brows, bcols, crows, ccols
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
! количество строк и столбцов матрицы A
      arows = 10
      acols = 20
! количество строк и столбцов матрицы B
      brows = 20
      bcols = 10
! количество строк и столбцов матрицы C
      crows = arows
      ccols = bcols
     if ( myid .eq. 0 ) then
! код главного процесса
! инициализация A и B
     do j = 1, acols
       do i = 1, arows
          a(i,j) = i
       enddo
     enddo
     do j = 1, bcols
       do i = 1, brows
          b(i,j) = i
        enddo
     enddo
! посылка матрицы B каждому подчиненному процессу
```

```
     do 25 i = 1,bcols
       call MPI_BCAST(b(1,i), brows, MPI_DOUBLE_PRECISION, &
       master, MPI_COMM_WORLD, ierr)
     enddo
       numsent = 0
! посылка строки каждому подчиненному процессу;

! в TAG - номер строки = i для простоты полагаем arows   >=
 ! numprocs-1 - 1
       do i = 1,numprocs-1
         do j = 1, acols
           buffer(j) = a(i,j)
         enddo
           call MPI_SEND (buffer, acols, &
           MPI_DOUBLE_PRECISION, i, i, MPI_COMM_WORLD, ierr)
           numsent = numsent+1
        enddo
           do i = 1, crows
             call MPI_RECV(ans, ccols, MPI_DOUBLE_PRECISION, &
             MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
             status, ierr)
             sender = status (MPI_SOURCE)
             anstype = status (MPI_TAG)
            do j = 1, ccols
               c(anstype, j) = ans(j)
            enddo
            if (numsent .lt. arows) then
            do j = 1, acols
              buffer(j) = a(numsent+1,j)
            enddo
              call MPI_SEND (buffer, acols,MPI_DOUBLE_PRECISION, &
              sender, numsent+1, MPI_COMM_WORLD, ierr)
              numsent = numsent+1
            else
! посылка признака конца работы
              call MPI_SEND(MPI_BOTTOM, 1, MPI_DOUBLE_PRECISION, &
              sender, 0, MPI_COMM_WORLD, ierr)
             endif
           enddo
```

```
        else
! код починенного процесса
! прием матрицы B каждым подчиненным
!процессом
        do i = 1,bcols
           call MPI_BCAST(b(1,i), brows, MPI_DOUBLE_PRECISION, &
           master, MPI_COMM_WORLD, ierr)
        enddo
! прием строки матрицы A каждым подчиненным процессом
 90       call MPI_RECV (buffer, acols, MPI_DOUBLE_PRECISION, &
          master, MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
        if (status (MPI_TAG) .eq. 0) then
           go to 200
        else
           row = status (MPI_TAG)
         do i = 1,bcols
           ans(i) = 0.0
          do j = 1, acols
 ! вычисление результатов
             ans(i) = ans(i) + buffer(j)*b(j,i)
          enddo
! посылка результата
          call MPI_SEND(ans, bcols, MPI_DOUBLE_PRECISION, &
          master, row, MPI_COMM_WORLD, ierr)
          go to 90
          endif
  200  continue
      endif
        call MPI_FINALIZE(ierr)
      stop
    end
```

11.4. Параллельные алгоритмы решения дифференциальных уравнений в частных производных

При изложении материала этого параграфа будем ссылаться на сведения, приведенные в Главе 9.

11.4.1. Решение эллиптических уравнений

Рассмотрим методы решения уравнения Лапласа (9.79) в прямоугольной области. Введем в этой области $\Re = \{(x,y) : 0 \leqslant x \leqslant a,\ 0 \leqslant y \leqslant b\}$ прямоугольную сетку с постоянными шагами по x и y. Пусть на границах этой прямоугольной области заданы граничные условия $f(x,y)$ и $g(x,y)$. Граничные условия, которые задаются на границах области $\Re$ бывают трех видов – Дирихле (9.7), Неймана (9.8) и Робина (9.9).

Самый простой и известный метод решения эллиптических уравнений – метод Якоби. Напомним, что очередное k+1 приближение находится по предыдущей k итерации. Завершение итераций происходит в тот момент, когда разница между двумя очередными итерациями не станет меньше некоторого заданного значения. Точность расчетов можно оценить с помощбю одной из следующих норм:

1. $\|x\|_1 = \sum_{i=1}^{n} |x_i|$
2. $\|x\|_2 = (\sum_{i=1}^{n} |x_i|^2)^{\frac{1}{2}}$
3. $\|x\|_\infty = \max\limits_{\forall i} |x_i|$

Решение, полученное методом Якоби, равномерно сходится к решению задачи Лапласа. Погрешность имеет порядок $O(h^2)$.

$$u_{i,j}^{k+1} = 0.25(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}) \qquad (11.20)$$

Рассмотрим последовательную программу решения уравнения Лапласа методом Якоби. В данной программе использованы некоторые возможности, которые предоставляет Fortran 95. Запись вида $real4 = selected\_real\_kind(6,37)$ позволяет задать значение мантиссы вещественного числа обычной точности равное 37 битам, а экспотенциальную часть равной 6 битам.

Программа 11.3 Последовательная программа решения эллиптических уравнений методом Якоби

```
    module jacobi_module
      implicit none
      integer, parameter :: real4 = selected_real_kind(6,37)
      integer, parameter :: real8 = selected_real_kind(15,307)
      real(real8), dimension(:,:), allocatable         :: unew
      real(real8), dimension(:,:), allocatable, target :: u
 ! solution array
      real(real8) :: tol=1.d-4, gdel=1.0d0
      real(real4) :: start_time, end_time
      integer :: m, iter = 0
      public

      contains
    subroutine bc(u, m)
!   pde: laplacian u = 0;      0<=x<=1;  0<=y<=1

! B.C.(Граничные условия):   u(x,0)=sin(pi*x);

! u(x,1)=sin(pi*x)*exp(-pi); u(0,y)=u(1,y)=0

!Решение: u(x,y)=sin(pi*x)*exp(-pi*y)

      implicit none
      integer m, j
      real(8), dimension(0:m+1,0:m+1) :: u
      real(8), dimension(:,:), pointer :: c
      real(8), dimension(0:m+1) :: y0
      y0 = sin(3.141593*(/(j,j=0,m+1)/)/(m+1))
      u = 0.0d0
 ! at x=0,1; all y plus initialize interior
      u(:,  0) = y0                ! at y = 0; all x
      u(:,m+1) = y0*exp(-3.141593) ! at y = 1; all x
     return
    end subroutine bc
   end module jacobi_module

    program Jacobi
      use jacobi_module
      real(real8), dimension(:,:),pointer :: c, n, e, w, s
```

```
      write(*,*)'Enter matrix size, m:'
      read(*,*)m

      call cpu_time(start_time)
 ! start timer, measured in seconds

      allocate ( unew(m,m), u(0:m+1,0:m+1) )
 ! memory for unew, u
      c => u(1:m  ,1:m  ) ! i,j   Current/Central
  ! for 1<=i<=m; 1<=j<=m
      n => u(1:m  ,2:m+1) ! i  ,j+1  North (of Current)
      e => u(2:m+1,1:m  ) ! i+1,j    East  (of Current)
      w => u(0:m-1,1:m  ) ! i-1,j    West  (of Current)
      s => u(1:m,0:m-1)   ! i  ,j-1  South (of Current)

      call bc(u, m)       ! set up boundary values
    do while (gdel > tol) ! iterate until error below threshold
      iter = iter + 1     ! increment iteration counter
     if(iter > 5000) then
        write(*,*)'iteration terminated (exceeds 5000)'
       stop                    ! nonconvergent solution
     endif
      unew = ( n + e + w + s )*0.25  ! new solution
      gdel = maxval(dabs(unew-c))    ! find local max error
     if(mod(iter,10)==0) &
        write(*,"('iter,gdel:',i6,e12.4)")iter,gdel
      c = unew                  ! update interior u
    enddo
     call cpu_time(end_time)      ! stop timer
      print *,'total cpu time =',end_time - start_time,' x 1'
      print *,'stopped at iteration =',iter
      print *,'the maximum error =',gdel
      deallocate (unew, u)
    end program Jacobi
```

Решение имеет вид:

Используем технологию распараллеливания OpenMP для построения параллельной программы для решения уравнения Лапласа.

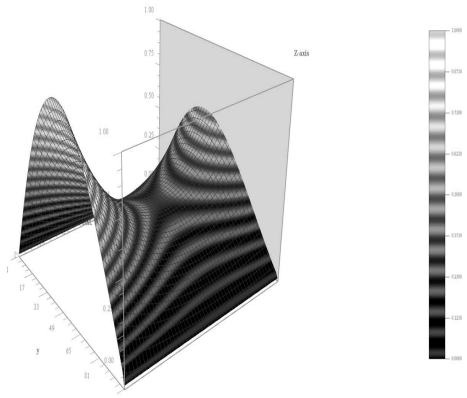

Рис. 11.1:

Программа 11.3 Параллельная программа решения эллиптических уравнений

```
module one
    integer,parameter:: n=4,maxiters=20
    real(8)::grid(n,n)=(/1,5,9,13,2,6,10,14,3,7,11,15,4,8,12,16/)
    real(8)::new(n,n)
  end module one

      program OMP_Fortran
      use one
! Jacobi iteration in Fortran 90 using OpenMP directives
      do i = 1,n
         write(*,'(4f9.4)') (grid(i,j),j=1,n)
       enddo
! read values for n and maxiters (not shown),

! the value of n includes the boundaries
       call jacobi()
       do i = 1,n
         write(*,'(4f9.4)') (grid(i,j),j=1,n)
       enddo
       read(*,*)
       stop
       end
```

```
    subroutine jacobi()
     use one
! allocate storage dynamically for the grids
       integer i,j,iters
       double precision maxdiff,tempdiff
! initialize boundaries of grid and new (not shown)

! initialize interior points to zeroes

!$omp parallel do
!$omp shared(n,grid,new), private(i,j)
      do j = 2,n-1
        do i = 2,n-1
          grid(i,j) = 0.0
          new(i,j) = 0.0
        enddo
      enddo
!$omp end parallel do

! initialize global variables
       iters = 1
       maxdiff = 0.0
! start main computational loop
!$omp parallel

!$omp& shared(n,maxiters,grid,new,iters)

!$omp& private(i,j,tempdiff)
!$omp& reduction(max: maxdiff)
       do while (iters.le.maxiters)
! update points in new
!$omp do
         do j = 2,n-1
           do i = 2,n-1
             new(i,j) = (grid(i-1,j) + grid(i+1,j) + &
             grid(i,j-1) + grid(i,j+1)) * 0.25
           enddo
         enddo
```

```
!$omp end do

! update points in grid
!$omp do
         do j = 2,n-1
           do i = 2,n-1
             grid(i,j) = (new(i-1,j) + new(i+1,j) + &
             new(i,j-1) + new(i,j+1)) * 0.25
           enddo
         enddo
!$omp end do

! one process updates the global iteration count
!$omp single
       iters = iters+2
!$omp end single

       enddo
! end of main computational loop

! compute maximum difference into a reduction variable

!$omp do
         do j = 2,n-1
           do i = 2,n-1
             tempdiff = abs(grid(i,j)-new(i,j))
             maxdiff = max(maxdiff,tempdiff)
               mydiff = tempdiff
           enddo
         enddo
!$omp end parallel do

!$omp end parallel
    end
```

Метод Якоби один из самых медленных методов решения эллиптических уравнений. Его применение оправдано только в учебных целях, так как он достаточно простой в реализации. Методы Гаусса - Зейделя и SOR (последовательной сверхрелаксации) сходятся быстрее, чем метод Якоби

и занимает почти в два раза меньше места в памяти. Несмотря на это построить параллельный алгоритм для этих методов с использованием технологии MPI не так просто. Рассмотрим еще один алгоритм. Мысленно окрасим точки расчетной области в два цвета - белый и черный как клетки шахматной доски. Начав с левой верхней точки "окрасим" точки через одну черным цветом, "покрасив" другие точки белым цветом. Такой метод называется методом "белое - черное". Иногда его называют методом "красное - черное".

Предложенный алгоритм обладает следующим свойством: у белых узлов соседними являются черные узлы, а у черных узлов - белые. Таким образом можно независимо и параллельно расчитывать черные и белые узлы. После каждого цикла расчета черных и белых узлов необходимо производить операцию обновления всех расчетных узлов. Граничные условия задаются в подпрограмме bc [36]. В параллельном коде Jacobi используются директивы MPI для нахождения числа доступных процессоров. Так как при решении данной задачи используется декартовая топология вычислительной сети, необходимо использовать MPI_Cart_Shift для обеспечения передачи информации.

Подпрограмма

```
subroutine update_bc_2
```

используется для модернизации черных ячеек сетки и одновременно использует MPI процедуру Sendrecv для парного приема и отправления информации. Подпрограмма

```
subroutine update_bc_1
```

используется для обновления красных узлов сетки. Подпрограмма

```
subroutine print_mesh
```

предназначена для вывода результатов в том случае, когда размер сетки не очень большой, меньше 10x10 ячеек.

Массивы указатели

```
c = news(v, m, mp, 0, 0)         ! i+0,j+0 center
n = news(v, m, mp, 0, 1)         ! i+0,j+1  north
e = news(v, m, mp, 1, 0)         ! i+1,j+0  east
w = news(v, m, mp,-1, 0)         ! i-1,j+0  west
s = news(v, m, mp, 0,-1)         ! i+0,j-1  south
```

используются для связи с различными частями пространства решения u. Они помогают также избежать потерь памяти при работе с большими массивами данных.

Директива $MPI\_Allreduce$ используется, для того, чтобы собрать глобальную ошибку от всех участвующих в вычислениях процессов, а также для того, чтобы определить, требуется ли дальнейшие итерации. Директива $MPI\_Allreduce$ требует много времени для выполнения, поэтому она вызывается только через каждые 100 итераций. Количество итераций, через которые необходимо вызывать $MPI\_Allreduce$ зависит от размера сетки и от требуемой точности.

Программа 11.4 Параллельная программа решения эллиптических уравнений

```
module jacobi_module
      implicit none
      integer, parameter :: real4 = selected_real_kind(6,37)
      integer, parameter :: real8 = selected_real_kind(15,307)
      real(real8), dimension(:,:), allocatable         :: vnew
      real(real8), dimension(:,:), allocatable, target :: v
! solution array
      real(real8) :: tol=1.d-4, del, gdel=1.0d0
      real(real4) :: start_time, end_time
! include "mpif.h"
! this brings in pre-defined mpi constants,...
      integer :: p, ierr, below, above, k, m, mp, iter=0

      contains
    subroutine bc(v, m, mp, k, p)
! pde: laplacian u = 0;      0<=x<=1;  0<=y<=1

! b.c.: u(x,0)=sin(pi*x); u(x,1)=sin(pi*x)*exp(-pi);
!u(0,y)=u(1,y)=0 ! solution: u(x,y)=sin(pi*x)*exp(-pi*y)

        implicit none
        integer m, mp, k, p, j
        real(real8), dimension(0:m+1,0:mp+1) :: v
        real(real8), dimension(:,:), pointer :: c
        real(real8), dimension(0:m+1) :: y0
        y0 = sin(3.141593*(/(j,j=0,m+1)/)/(m+1))
      if( p > 1 ) then
```

```
        u = 0.0
       if (k == 0  ) v(:,   0) = y0
       if (k == p-1) v(:,mp+1) = y0*exp(-3.141593)
      else
        v = 0.0
        v(:,0) = y0
        v(:,m+1) = y0*exp(-3.141593)
      end if
     return
    end subroutine bc

    subroutine borders(k, below, above, p)
      implicit none
      integer :: k, below, above, p
    if(k == 0) then
      below  = -1  ! tells mpi not to perform send/recv
      above = k+1
    else if(k == p-1) then
      below  = k-1
      above = -1   ! tells mpi not to perform send/recv
    else
      below  = k-1
      above = k+1
    endif
    return
  end subroutine borders

    subroutine update_bc_2( v, m, mp, k, below, above )
      include "mpif.h"
      integer :: m, mp, k, below, above, ierr
      real(real8), dimension(0:m+1,0:mp+1) :: v
      integer status(mpi_status_size)

     call mpi_sendrecv(v(1,mp), m, mpi_double_precision, &
          above, 0, v(1,   0), m, mpi_double_precision,  &
          below, 0, mpi_comm_world,status, ierr )
     call mpi_sendrecv(v(1,1), m, mpi_double_precision,  &
          below, 1, mpi_double_precision,  below, 1,     &
          v(1,mp+1),m,mpi_double_precision, above, 1,    &
```

```
          mpi_comm_world, status, ierr)
      return
    end subroutine update_bc_2


    subroutine update_bc_1(v, m, mp, k, below, above)
      implicit none
      include 'mpif.h'
      integer :: m, mp, k, ierr, below, above
      real(real8), dimension(0:m+1,0:mp+1) :: v
      integer status(mpi_status_size)
! select 2nd index for domain decomposition to have stride 1

! use odd/even scheme to reduce contention in message passing
     if(mod(k,2) == 0) then     ! even numbered processes
        call mpi_send( v(1,mp), m, mpi_double_precision,  &
             above, 0, mpi_comm_world, ierr)
        call mpi_recv( v(1,0), m, mpi_double_precision,   &
             below, 0, mpi_comm_world, status, ierr)
        call mpi_send( v(1,1), m, mpi_double_precision,   &
             below, 1, mpi_comm_world, ierr)
        call mpi_recv( v(1,mp+1), m, mpi_double_precision,&
         above, 1, mpi_comm_world, status, ierr)
    else                         ! odd numbered processes
      call mpi_recv( v(1,0   ), m, mpi_double_precision,  &
           below, 0, mpi_comm_world, status, ierr)
      call mpi_send( v(1,mp  ), m, mpi_double_precision,  &
           above, 0, mpi_comm_world, ierr)
      call mpi_recv( v(1,mp+1), m, mpi_double_precision,  &
           above, 1, mpi_comm_world, status, ierr)
      call mpi_send( v(1,1   ), m, mpi_double_precision,   &
           below, 1, mpi_comm_world, ierr)
     endif
     return
    end subroutine update_bc_1

    subroutine print_mesh(v,m,mp,k,iter)
      implicit none
      integer :: m, mp, k, iter, i, out
```

```
  real(real8), dimension(0:m+1,0:mp+1) :: v
    out = 20 + k
   do i=0,m+1
     write(out,"(2i3,i5,' => ',4f10.3)")k,i,iter,v(i,:)
   enddo
     write(out,*)'++++++++++++++++++++++++++'
 return
 end subroutine print_mesh
end module jacobi_module

module types_module
  implicit none
  integer, parameter::real4=selected_real_kind(6,37)
  integer, parameter::real8=selected_real_kind(15,307)
  public
end module types_module

module sor_module
  use jacobi_module
  type oddeven
  real(real8), dimension(:,:), pointer :: odd
  real(real8), dimension(:,:), pointer :: even
  end type oddeven
  type whiteblack
  type (oddeven) :: white, black
  end type whiteblack
  real(real8), parameter :: pi=3.141593d0
  real(real8) :: omega, rhoj, rhojsq, delr, delb
  contains

function news(v, m, mp, i, j) integer :: m, mp, i, j
  real(real8), dimension(0:m+1,0:mp+1), target :: v
  type (whiteblack) :: news news%black%odd => &
                       v(i+1:i+m:2,j+2:j+mp:2)
  news%black%even => v(i+2:i+m:2,j+1:j+mp:2)  &
                     news%white%odd => &
  v(i+1:i+m:2,j+1:j+mp:2) news%white%even =>  &
                     v(i+2:i+m:2,j+2:j+mp:2)
end function news
```

```
    subroutine update_u(c, n, e, w, s, vnew, m, mp, omega, del)
      implicit none
      integer m, mp
      real(real8) :: omega, del
      real(real8), dimension(1:m/2,1:mp/2) :: vnew
      type (oddeven) :: c, n, e, w, s
      vnew = (n%odd + e%odd + w%odd + s%odd )*0.25d0 c%odd =&
      (1.0d0 - omega)*c%odd + omega*vnew
      del = sum(dabs(vnew-c%odd)) vnew = (n%even + e%even + &
      w%even + s%even)*0.25d0 c%even=(1.0d0 - omega)*c%even+&
      omega*vnew
      del = del + sum(dabs(vnew-c%even))
    end subroutine update_u
    end module sor_module

        program sor_cart
      use types_module
      use sor_module
      type (whiteblack) :: c, n, e, w, s
      integer, parameter :: period=0, ndim=1
      integer :: grid_comm, me, iv, coord, dims
      logical, parameter :: reorder = .true.
! starts mpi
      call mpi_init(ierr)
! get current process id
      call mpi_comm_rank(mpi_comm_world, k, ierr)
! get # procs from env or command line
      call mpi_comm_size(mpi_comm_world, p, ierr)

     if(k == 0) then
        write(*,*)'enter matrix size, m :'
        read(*,*)m
     endif
        call mpi_bcast(m, 1, mpi_integer, 0, &
             mpi_comm_world, ierr)

        rhoj = 1.0d0 - pi*pi*0.5/(m+2)**2
        rhojsq = rhoj*rhoj
```

```
        mp = m/p
        allocate (vnew(m/2,mp/2), v(0:m+1,0:mp+1))
!start timer, measured in seconds
        call cpu_time(start_time)

  ! create 1d cartesian topology for matrix
        dims = p
        call mpi_cart_create(mpi_comm_world, &
             ndim, dims, period, reorder, grid_comm, ierr)
        call mpi_comm_rank(grid_comm, me, ierr)
        call mpi_cart_coords(grid_comm, me, &
             ndim, coord, ierr)
        iv = coord
! set up boundary conditions
        call bc(v, m, mp, iv, p)

        call mpi_cart_shift(grid_comm, 0, 1, below, &
             above, ierr)

        c = news(v, m, mp, 0, 0)         ! i+0,j+0  center
        n = news(v, m, mp, 0, 1)         ! i+0,j+1  north
        e = news(v, m, mp, 1, 0)         ! i+1,j+0  east
        w = news(v, m, mp,-1, 0)         ! i-1,j+0  west
        s = news(v, m, mp, 0,-1)         ! i+0,j-1  south

        omega = 1.0d0
! update white
        call update_u(c%white, n%white, e%white, &
        w%white, s%white, vnew, m, mp, omega, delr)
        call update_bc_2( v, m, mp, iv, below, above)
        omega = 1.0d0/(1.0d0 - 0.50d0*rhojsq)
        call update_u(c%black, n%black, e%black, &
! update black
        w%black, s%black, vnew, m, mp, omega, delb)
        call update_bc_2( v, m, mp, iv, below, above)
      do while (gdel > tol)
        iter = iter + 1    ! increment iteration counter
        omega = 1.0d0/(1.0d0 - 0.25d0*rhojsq*omega)
! update white
```

```
        call update_u(c%white, n%white, e%white, w%white, &
        s%white, vnew, m, mp, omega, delr)
        call update_bc_2( v, m, mp, iv, below, above)
        omega = 1.0d0/(1.0d0 - 0.25d0*rhojsq*omega)
! update black
        call update_u(c%black, n%black, e%black, w%black, &
        s%black, vnew, m, mp, omega, delb)
        del = (delr + delb)*4.d0
      if(mod(iter,100)==0) then
         del = (delr + delb)*4.d0
         call mpi_allreduce( del, gdel, 1, &
         mpi_double_precision, mpi_max,    &
         mpi_comm_world, ierr )  ! find global max error
      if(k == 0) write(*,'(i5,3d13.5)')iter,del,gdel,omega
      endif
     enddo

      call cpu_time(end_time)          ! stop timer

     if(k == 0) then
        print *,'########################################'
        print *,'total cpu time =',end_time-start_time,' x',p
        print *,'stopped at iteration =',iter
        print *,'the maximum error =',del
        write(40,"(3i5)")m,mp,p
     endif
        write(41+k,"(6d13.4)")v
        deallocate (vnew, v)

        call mpi_finalize(ierr)

    end program sor_cart
```

Литература

[1] Д. Коханер, Л. Моулер, С. Нэш. Численные методы и программное обеспечение. М., Мир, 2001 г., 575 стр.

[2] Д. Мак-Кракен, У. Дорн. Численные методы и программирование на Фортране. М., Мир, 1977, 584 стр.

[3] А.А. Самарский, А.П. Михайлов. Математическое моделирование. Идеи, Методы, примеры. М., Наука, 1997 г., 320 стр.

[4] А.А. Самарский. Введение в численные методы. М., Наука, 1977 г.

[5] А.А. Самарский, Ю.П. Попов. Разностные схемы газовой динамики. М., Наука, 1982 г.

[6] В.В. Воеводин, Вл.В. Воеводин. Параллельные вычисления. Bhv, С.-Петербург, 2004 г., 599 стр.

[7] Д.Э. Кнут. Искусство программирования. М., Мир, 2000 г. 753 стр.

[8] Крис Х. Паппас, Уильм Х. Мюррей III. Отладка в C++. Бином, 2001 г., 507 стр.

[9] Д. Ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ. М., Мир, 1985 г., 332 стр.

[10] А.Н. Тихонов, А.А. Самарский. Уравнения математической физики. М., Наука, 1977 г., 735 стр.

[11] В.М. Головизнин, И.М. Кобринский, А.В. Соловьев, Е.В. Соловьева. Комплекс обучающих и методических программ для численного решения уравнений конвективного переноса.

[12] А.В. Попов. Практикум на ЭВМ: Разностные методы решения квазилинейных уравнений первого порядка. Часть 1. Ж. Вычислительные методы и программирование, 2003 г., Т. 4, стр. 16 - 27.

[13] Ю.И. Рыжиков. Современный Фортран. Учебник. СПб., Корона, 2004 г., 288 стр.

[14] Н.С. Бахвалов, Н.П. Жидков, Г.М. Кобельков. Численные методы. – М. Наука, 1983.

[15] В.И. Киреев, А.В. Пантелеев. Численные методы в примерах и задачах. Высшая школа, 2006, 480 стр.

[16] О.В. Бартеньев. ФОРТРАН для профессионалов. Диалог МИФИ, 2000, т.1, 448 стр.

[17] Д.Г. Мэтьюз, К.Д. Финк. Численные методы. Использование MATLAB. Москва, Вильямс, 2001, 713 стр.

[18] Д. Андерсен, Дж. Таннехилл, Р. Плетчер. Вычислительная гидродинамика и теплообмен. М., Мир, 1990 г., в 2 т.

[19] К. Флетчер. Вычислительные методы в динамике жидкостей. М., Мир, 1991, в 2 т.

[20] Э. Оран, Дж.Борис. Численное моделирование реагирующих потоков. М., Мир, 1990 г., 661 стр.

[21] Гущин В.А., Коньшин В.Н Численное моделирование волновых движений жидкости. Сообщения по прикладной математике Препринт ВЦ АН СССР. 1985. 36 с

[22] Bovrel M., Montagne J.L Numerical study of a non - centered scheme with application to aerodynamics AIAA Paper. 1985. 1;. 85 - 1497. [Idem, in AIAA 7th Comput. Fluid Dyn. Conf. Cincinnati, Ohio, 1985, July 15 - 17. A Collect. Techn. Papers, 88 - 97, AIAA, New York]

[23] Воробьев О.В., Холодов А.С Об одном методе численного интегрирования одномерных задач газовой динамики Математическое моделирование. 1996. т. 8. 1;1. С. 77 - 92

[24] Yih Nen Jeng, Uon Jan Payne. An Adaptive TVD Limiter. J. of Comp. Physics, 118, 229-241, (1995), pp. 229-241.

[25] A. Harten, S. Osher, Uniformly high-order accurate nonoscillatory schemes, I, SIAM J. Numer.Anal.24,279 (1987).

[26] Г.К.Эндрюс. Основы многопоточного, параллельного и распределенного программирования. М., Вильямс, 2003, 512 стр.

[27] OpenMP. Fortran Application Program Interface. Oct 1997 1.0

[28] В.Д. Корнеев. Параллельное программирование в MPI. Москва, Ижевск. 2003, 304 стр.

[29] А.С. Антонов. Параллельное программирование с использованием технологии MPI. Издательство МГУ им.М.В. Ломоносова, НИВЦ, 2004 г. 72 стр.

[30] А. А. Букатов, В. Н. Дацюк, А. И. Жегуло. Программирование многопроцессорных вычислительных систем, Ростов-на-Дону, 2003, 208 стр.

[31] OpenMP Application Program Interface, Version 2.5 May 2005, Copyright © 1997-2005 OpenMP Architecture Review Board.

[32] Введение в параллельные и векторные методы решения параллельных систем. Москва, Мир 1991, 365 с.

[33] Bertsekas, D.P., Tsitsiklis, J.N. (1989). Parallel and distributed Computation. Numerical Methods. - Prentice Hall, Englewood Cliffs, New Jersey.

[34] В.П. Гергель. Теория и практика параллельных вычислений. Бином, 2007 г., 423 стр.

[35] Г.И. Шпаковский, Н.И. Серикова. Программирование для многопроцессорных систем в стандарте MPI. Минск, БГУ, 2002, 323 с.

[36] Parallel Successive Over Relaxation Red-black Scheme. www.scv.bu.edu/documentation/tutorials/MPI