Шамин Роман Вячеславович

Численные методы и математическое моделирование на C#

Москва — 2016

Оглавление

Введение

Настоящий курс, посвящен объектно-ориентированному изложению современных численных методов на языке C# . Идея данного курса состоит в том, чтобы показать мощность объектно-ориентированного подхода при программировании задач научного программирования. Численные методы представляют собой классическую область для применения вычислительной техники. До появления первых вычислительных средств теория численных методов обгоняла вычислительные возможности, однако со стремительной эволюцией компьютеров и быстрым развитием технологий программирования ситуация поменялась. Начиная с некоторых пор уже вычислительные возможности обгоняют теоретические успехи численных методов. Конечно, при проведении промышленных расчетов, как правило, применяются самые современные технологии, однако учебники по численным методам часто передают дух предыдущей эпохи. Более того, во многих классических учебниках вопросам программирования численных методов вообще не уделяется места. Понимаю важность изложения теории численных методов, нельзя не признать, что вопросы реализации численных методов являются отнюдь не техническими. Настоящий курс ставит своей целью показать, как используя современный язык программирования C# и объектно-ориентированный подход, можно реализовать некоторые наиболее часто используемые численные методы.

Необходимо отметить, что данный курс не является учебником по

численным методам — мы не рассматриваем достаточное многообразие вычислительных методов и совсем не приводим никаких доказательств. Также наш курс не является и пособием по программированию на C# . В данном курсе мы рассматриваем реализацию на C# с использованием объектно-ориентированный подхода некоторых численных методов для решения различных задач. Практически каждая тема излагается по следующей схеме — сначала идет сжатое теоретическое изложение данной темы, потом приводятся примеры для конкретных постановок вычислительных задач. После теоретического изложение мы переходим к практической реализации. Как правило, реализация начинается с построения абстрактных классов, а потом их реализация для конкретных проблем. Обязательно приведенные листинги заканчиваются вычислительными экспериментами, которые должны продемонстрировать теоретические выкладки. В нашем курсе мы хотим дать возможность «потрогать» с помощью программирования те или иные понятия математики. Темы, которые мы затрагиваем довольно широки: конструктивные действительные числа, функциональные пространства (метрические, банаховы, гильбертовы), операторные уравнения, дифференциальные уравнения, управляемые системы, динамические системы, теория игр, агентное моделирование и др. Последняя лекция несколько выбивается из общего стиля изложения. В этой лекции мы приводим результаты одно из реального вычислительного эксперимента по моделированию волн-убийц в Мировом океане. Этой демонстрационной лекцией мы хотим показать как проходят настоящие масштабные вычислительные эксперименты.

Для понимания данного курса необходимы следующие базовые знания. Во-первых, основы программирования на языке C# . Предполагается, что слушатель знает о базовых конструкциях этого языка. Используемые конструкции объектно-ориентированного программирования мы подробно рассматриваем. Во-вторых, от предполага-

ется, что слушатель владеет математикой в объеме первых двух курсов технических университетов. Необходимо знать основы математического анализа, линейной алгебры, основы обыкновенных дифференциальных уравнений. Знания из функционального анализа не предполагаются — все необходимые понятия мы вводим по ходу текста. Также не требуется предварительных знаний в теории алгоритмов, в теории управления, динамических систем, теории игр и т.д. Правда, как уже отмечалось, в последней версии мы используем сложные понятия дифференциальных уравнений в частных производных и теории нелинейных волн, для понимания которых необходимы специальные знания. Однако эта демонстрационная лекция и ее можно воспринимать «как узор».

Благодарности. Идея данного курса многократно обсуждалась с моим другом — доцентом, кандидатом физ.-мат. наук Дмитрием Сошниковым, которому автор выражает искреннюю благодарность.

Также я благодарен своим студентам, которым я читаю лекции, и чья активная позиция на лекциях помогла улучшить данный курс.

Часть I

Объектно-ориентированнная реализация математических объектов

Глава I

Принципы объектно-ориентированного программирования

1. Концепции современного программирования

Изначально программирование возникло как математическая дисциплина и первым применением новой области знания стало решение естественно-научных задач. Не будет большой ошибкой сказать, что первые программируемые вычислительные машины возникли в ходе создания ядерного оружия в США и СССР. При этом первыми программистами были математики, физики, электронщики. На первоначальном этапе программирование было наукой, искусством, развлечением, но никак не ремеслом. Основными проблемами, стоявшими перед программистами были мизерные (по современным меркам) объемы оперативной памяти и низкое быстродействие компьютеров того времени. С расширением сферы программирования стали возникать проблемы с организацией процесса программирования. После того как программное обеспечение стало программным продуктом, то вопросы технологии программирования стали выходить на первый план.

Мы рассмотрим историю различных парадигм программирования. Первоначально программирование начиналось без каких-либо единых подходов, поскольку программы писались в машинных кодах и программирование сильно зависело от конкретной машины. Однако с развитием вычислительной техники и с построением серийных компьютеров возникли универсальные языки программирования высокого уровня. Первым известным языком программирования был язык Фортран. Этот язык, название которого является сокращением от «транслатор формул», представлял собой уже в определенном смысле новую парадигму программирования, которую мы назовем **парадигмой языков высокого уровня**. Программирование на языке Фортран представляет собой написание программ с использованием формализованных команд, которые в определенном смысле можно считать математическим текстом. Конечно, для многих классических математиков некоторые команды вызывали удивление и непонимание. Например команда присвоения:

$$x = x + 1$$

вызывала недоумение. Забегая вперед, заметим, что Н. Вирт при создании языка Паскаль операция присвоения обозначена через «:=». Тем не менее с языка Фортран началось алгоритмическое программирование, когда появился более-менее стандартный язык для написания программ. При этом стало возможным писать программы «впрок», когда при создании программ можно было использовать ранее созданные тексты программ. Многие часто используемые подпрограммы стали оформляться в виде специальных библиотек. С дальнейшим развитием компьютерной техники и расширением сферы программирования возникла следующая парадигма программирования, которую мы назовем **«парадигма процедурного программирования»**. Процедурное программирование основано на том, что при создании больших программных продуктов необходимо раз-

бивать всю программу на отдельные подпрограммы (процедуры) и использовать написанные ранее подпрограммы, оформленные в виде подключаемых библиотек. Именно с этого момента стало возможным создания сложных программных комплексов. Если первоначально программы писались одним программистом в каждый момент времени, то при создании сложных программных комплексов необходимо привлекать целый коллектив программистов, которые работали одновременно. Это потребовало изменений в организацию процесса программирования. Процедурное программирование позволяло разделить процесс программирования на отдельные задачи, которые можно было выполнять одновременно разными программистами. При этом вместо написания большой программы, создавались относительно небольшие подпрограммы, что позволяло значительно легче отлаживать программы и модернизировать ранее созданные комплексы.

Если на начальном этапе развития программирования схема создания программы была следующая:

1. Построение алгоритма и его формальное описание
2. Программирование (кодирование алгоритма)
3. Отладка программы
4. Использование программы

то с развитием процедурного подхода появился еще один важный этап — *проектирование*. С появлением этапа проектирования программы становятся программными комплексами. Развитое процедурное программирование становится новой парадигмой программирования, которая называется **парадигмой структурного программирования**.

Под структурным программированием мы будем понимать такой метод разработки программ, который начинается с тщательного

проектирования структуры программы методом «сверху вниз». При этом каждый блок должен быть спроектирован таким образом, чтобы его создание и отладка могла быть автономной. Часто под структурным программированием понимают и использование «правил хорошего тона», например исключение оператором GOTO, а также подробное документирование текстов программ. Идеология структурного программирование имеет своих основных авторов, к которым относятся такие известные ученые, как Э.В.Дейкстра и Н.Вирт.

Следующая парадигма программирования — **«парадигма объектно-ориентированного программирования»**. Конечно, слово «следующая» нужно понимать не в хронологическом порядке, а по распространению. Поскольку, если структурное программирование было создано в 70-х годах, то объектно-ориентированное программирование появилось в языке Simula, в 1967 году. Основы объектно-ориентированного программирования в языке C# мы рассмотрим подробно в нашем курсе в других главах. Объектно-ориентированное программирование явилось новым шагом в технологии программирования и позволило создавать сложные программные комплексы. В настоящее время подход объектно-ориентированнго программирования является стандартом для промышленного программирования. Заметим, что, как правило, для реализации объектно-ориентированного подхода необходима соответствующая поддержка со стороны языковых конструкций. К числу наиболее распространенных объектно-ориентированнных языков относятся:

- C++
- Java
- C#
- Delphi
- VB.NET

- Perl
- PHP
- и многие другие.

Заметим, что объектно-ориентированный подходоказал большое влияние на архитектуру операционной системы Microsoft Windows. Хотя в начале основным средством для программирования под Windows был язык C, понятие «окна» в Windows вполне соответствовало понятию объекта или класса.

Любая программа — это обработка данных, однако часто программы создается для обработки не просто данных, а данных описывающих те или иные объекты (персоны, машины, математические функции и т.д.). Понятие «объекта» включает в себя некоторую информацию, описывающую состояние объекта (при этом эта информация может быть видимой из вне объекта, а может быть и невидимой изнутри объекта). Далее объект может уметь выполнять определенные команды, которые поступают из внешней среды по строго установленному интерфейсу. В объектно-ориентированном программировании это называется **инкапсуляцией**. Еще процедурное программирование учит, что необходимо минимизировать повторяющиеся куски программы. Необходимо писать такие программы, которые будут удовлетворять требованиям повторного использования кода. В объектно-ориентированном программировании для этого используются механизмы **наследования** и **полиморфизма**.

Хотя объектов может быть довольно много, но часто рассматриваемые объекты имеют общие черты. Для того, чтобы не создавать повторяющегося кода при описании объектов, был придуман механизм наследования. Допустим, что мы моделируем транспортную сеть, в которая состоит из автобусов, маршрутных такси, трамваев и троллейбусов. Соответственно, у нас должно быть четыре объек-

та: автобус, такси, трамвай и автобус. Каждый объект характеризуется рядом показателей: положение, скорость, количество пассажиров, возможность проехать по выбранному маршруту и т.д. Ясно, что некоторые характеристики присущи всем четырем объектам, далее часть характеристик (возможность проехать по данной улице) присущи только трамваю и троллейбусу потому, что для этого транспорта необходимо заранее проложить провода, а для трамвая и рельсы. Покажем, как такие объекты следует реализовать на основе объектно-ориентированный подхода. Первым делом введем объект «траспортное средство», все наши объекты будут наследниками от этого объекта. Далее введем объекты: «автотраспорт» и «электротранспорт». Эти объекты будут наследниками от объекта «траспортное средство». Наконец, введем объекты: «автобус», «такси», «трамвай» и «троллейбус». При этом первые два объекта наследники от объекта «автотраспорт», а последние два от объекта «электротранспорт». При наследовании объекта на автоматически будет доступен код, который мы создавали для родительского класса. Технология полиморфизма позволяет, грубо говоря, модифицировать созданный ранее код для наследуемого класса. В следующей главе мы подробно рассмотрим объектно-ориентированное программирование на примере языка C#.

А что после объектно-ориентированного программирования? В последнее время набирает популярность логическое и функциональное программирование. Функциональное программирование довольно радикально отличается от рассмотренных ранее парадигм программирования. Если до рассматриваемые до этого парадигмы программирования были основаны на императивном программировании, то есть по шаговым заданием операций, то функциональное программирование основано на вычислениях функций. В программах, написанных с помощью функционального программирования, нет состояния программы в каждый момент времени. Считается, что функцио-

нальное программирование намного ближе к математическому мышлению. Действительно, функциональное программирование основано на λ-исчислении, известном в математической логики, посвященной теории вычислимости. Однако, большинство численных методов, применяемых на практике основаны на итерационных процедурах, поэтому пока функциональное программирование находит лишь ограниченное применение в вычислительной математике. Среди языков функционального программирования отметим:

- LISP
- Haskell
- ML
- F#

Интересно, что язык LISP был разработан в 1958 году, что раньше таких известных процедурных языков, как ALGOL-60, Pascal, C... Однако именно сейчас увеличился интерес функциональным языкам программирования, что подтверждается поддержкой языка F# в Microsoft Visual Studio 2010. Наконец отметим, что по выражению Дмитрия Сошникова функциональное программирования позволяет больше думать и меньше писать.

Ключевые термины

Парадигма объектно-ориентированного программирования — использование объектов и классов, а также технологий инкапсуляции, наследования и полиморфизма.

Парадигма процедурного программирования — систематическое использование отдельных небольших модулей (подпрограмм или процедур.

Парадигма структурного программирования — метод разработки программ «сверху вниз», использование специальных правил хорошего тона при программировании.

Парадигма языков высокого уровня — использование машинно-независимых языков программирования, запись программ с помощью операторов языка программирования а не машинных команд.

Краткие итоги: Различные парадигмы программирования сменяются со временем, следуя за прогрессом вычислительной техники. Популярность парадигм программирования не всегда совпадает со временем появления на свет.

2. Инкапсуляция

Начиная с этой лекции мы будем подробно рассматривать принципы объектно-ориентированнго программирования только на примере языка C#. Язык C# является полноценным объектно-ориентированным языком программирования со строгим контролем типов. Мы будем рассматривать объектно-ориентированное программирование только для языка C#, поскольку этот язык поддерживает все основные конструкции объектно-ориентированный подхода за исключением множественного наследования.

В C# объектно-ориентированное программирование поддерживается с помощью классов. **Класс** — это пользовательский тип данных, который включает в себя как данные, так и программный код в виде функций. Данные хранимые в классе называются полями, а функции класса называются методами. Также класс может содержать свойства и индексаторы, которые в определенном смысле объединяют поля и методы. После того, как класс описан, можно создавать переменные с типом этого класса. Эти переменные называются **экземплярами класса** или объектами данного класса. Приведем пример простейшего класса:

```
class TAirplan
{
    string Name;
```

```
        double Altitude;
        double MaxAltitude = 12000;

        void IncAltitude(double delta)
        {
            Altitude += delta;
            if(Altitude > MaxAltitude)
            {
                Altitude = MaxAltitude;
            }
        }
    }
```

Теперь у нас есть новый тип данных — TAirplan. Если мы попробуем использовать этот класс следующим образом:

```
TAirplan Airplan;

Airplan.Name = "Boeing"; Airplan.Altitude = 1000;
Airplan.IncAltitude(100);
```

то обнаружим следующие ошибки. Во-первых, для создания экземпляра класса недостаточно объявить соответствующую переменную. Экземпляр класса необходимо создать с помощью оператора New. Дело в том, что переменная типа класса на самом деле является не переменной, а ссылкой на переменную. По сути **ссылка** — это адрес в памяти, где должна храниться переменная. Поэтому процедура создания класса включает в себя выделение соответствующей памяти, инициализации полей класса и присвоение ссылке адреса выделенной памяти. Создать экземпляр нашего класса можно следующим образом:

```
TAirplan Airplan = new TAirplan();
```

Во-вторых, использование полей класса и вызов его методов в строках:

```
Airplan.Name = "Boeing"; Airplan.Altitude = 1000;
Airplan.IncAltitude(100);
```

будет невозможным. Дело в том, что по умолчанию определенные поля и методы являются закрытыми от внешнего обращения. К таким полям можно обращаться только из методов самого класса. Для того, чтобы сделать возможным общаться к полям и методам класса необходимо использовать **спецификаторы доступа**. В C# существуют следующие спецификаторы доступа:

- public
- protected
- private
- new
- internal
- static
- readonly
- volatile

Некоторые спецификаторы могут использоваться совместно, а некоторые являются взаимоисключающими. По умолчанию, если не указан спецификатор доступа, полям и методам присваивается спецификатор private. Этот спецификатор означает, что к данному полю и методу не возможно обратиться из другого класса и даже из класса — наследника нашего класса. Спецификатор private аналогичен спецификатору private за исключением того, что поля и методы с этим

спецификатором «видны» для классов наследников данного класса. Наиболее либеральным является спецификатор public. К полям и методам, помеченным этим спецификатором, можно обращаться из любых других классов. Может показаться, что легче всего помечать все поля и методы public, чтобы «не мучаться». Часто начинающие программисты именно так и поступают. Однако такой подход противоречит идеологии объектно-ориентированного программирования. Важная идея инкапсуляции состоит в сокрытии внутренних полей и методов от внешней среды. Может показаться удивительным, но порой чем меньше мы знаем о внутреннем устройстве класса, тем легче и безопаснее им пользоваться.

Возвращаясь к нашему примеру, разумно дать возможность вызывать метод IncAltitude() для изменения высоты. Для этого нужно назначит для метода IncAltitude спецификатор доступа — public. Но не стоит давать возможность пользователям класса самостоятельно изменять высоту путем доступа к полю Altitude, поскольку тогда возможно превышение порога высоты. В тоже время, если у поля Altitude будет спецификатор доступа private, то пользователь не сможет не только изменить это поле, но и прочитать. Для этого нужно создать еще один метод, который будет выдавать пользователю значение высоты:

```
public double GetAltitude()
{
    return Math.Round(Altitude);
}
```

Заметим, что таким образом мы контролируем, чтобы пользователь всегда получал значение высоты, округленное до ближайшего целого.

Однако если мы создадим наш класс в таком виде:

```
class TAirplan
```

```
{
    string Name;
    double Altitude;
    double MaxAltitude = 12000;

    void IncAltitude(double delta)
    {
        Altitude += delta;
        if(Altitude > MaxAltitude)
        {
            Altitude = MaxAltitude;
        }
    }

    public double GetAltitude()
    {
        return Math.Round(Altitude);
    }
}
```

и попробуем его использовать таким образом:

```
TAirplan Airplan = new TAirplan(); // создать объект
Airplan.IncAltitude(100); // увеличить высоту
Console.WriteLine("Высота = {0}", Airplan.GetAltitude());
// показать высоту
```

то мы увидим строку «Высота = 100». Однако есть одно обстоятельство, состоящее в том, что мы никак не инициализировали наши поля перед использованием. Конечно, мы можем знать, что C# всегда инициализирует переменные типа double нулем, но желательно инициализировать переменные явно. К счастью, в объектно-ориентированном программировании каждый класс может иметь специальный

метод (и даже не один) называемый конструктором. **Конструктор** — это метод, который всегда вызывается при создании экземпляра класса. Кстати, если в классе не определен ни один конструктор, то компилятор создать конструктор по умолчанию. Конструктор имеет следующие черты, которые выделяют его из других методов.

1. Имя конструктора всегда совпадает с именем класса. Никакой другой метод не может иметь имя, совпадающее с именем класса.

2. Конструктор не имеет возвращаемого типа. Никакой другой метод не может не иметь возвращаемого типа.

3. Конструктор можно вызвать только путем создания экземпляра класса.

Добавим в наш класс конструктор:

```
class TAirplan
{
    public readonly string Name;
    double Altitude;
    double MaxAltitude = 12000;

    public TAirplan(string Name, double Altitude)
    {
        this.Name = Name;
        this.Altitude = Altitude;
    }

    public void IncAltitude(double delta) { ... }

    public double GetAltitude() { ... }
}
```

Теперь создавать экземпляр нашего класса нужно следующим образом:

```
TAirplan Airplan = new TAirplan("Boeing", 0);
```

Заметим, что мы пометили поле Name спецификатором доступа readonly, для того, чтобы извне можно было читать это поле, но нельзя было модифицировать. Обратим Ваше внимание на ключевое слово this, для доступа к полям самого класса. Это необходимо для того, чтобы отделить переменные–параметры от полей класса.

Рассмотренная выше технология программирования называется инкапсуляцией. Инкапсуляцией называется технология программирования, при которой реализуется тщательный контроль доступа к внутренним полям и методам класса. На наш взгляд инкапсуляция является наиболее важным принципом объектно-ориентированного программирования.

Ключевые термины

Класс — пользовательский тип данных, который включает в себя как данные, так и программный код в виде функций.

Конструктор — метод, который всегда вызывается при создании экземпляра класса.

Спецификаторы доступа — ключевые слова языка C# для указания уровня доступа к полю или методу класса.

Ссылка — именованный адрес в памяти, где хранится переменная, для доступа к этой переменной.

Экземпляр класса — переменная типа класс.

Краткие итоги: Рассмотрена технология инкапсуляции на примере классов C# . На модельных примерах показано применение этой технологии и ее эффективность.

3. Наследование и полиморфизм

Чтобы продемонстрировать механизм наследования классов рассмотрим несколько нарочито простых классов, которые будут описывать транспортные средства: автобус, такси, троллейбус и трамвай. Базовым классом у нас будет класс «траспортное средство», который мы определим следующим образом:

```
class TTransport
{
    protected int Speed; // скорость
    protected int Massa; // Масса
    protected int Payload; // грузоподъемность

    public TTransport(int Speed, int Massa, int Payload)
    {
        this.Speed = Speed;
        this.Massa = Massa;
        this.Payload = Payload;
    }

    public void Print() // распечатать информацию
    {
        Console.WriteLine("Speed = {0}", Speed);
        Console.WriteLine("Massa = {0}", Massa);
        Console.WriteLine("Payload = {0}", Payload);
    }
}
```

Смысл этого класса хранить и распечатывать информации о транспортном средстве. Любое транспортное средство имеет такие характеристики как скорость, масса и грузоподъемность. Однако для автотранспорта есть еще дополнительные характеристики такие, как

расход топлива, а у электротранспорта есть такая дополнительная характеристика как напряжение. Поэтому мы создадим еще два класса. С использованием технологии наследования. **Наследование** — это технология проектирования классов, позволяющая создавать новые классы на базе предыдущих классов. При этом классы наследники будут иметь все поля и методы родительского класса. А к тем полям и методам, которые имеют спецификатор доступа public или protected, можно будет обращаться из наследного класса. Приведем определения этих классов:

```
class TAuto : TTransport
{
    protected int Petrol; // Расход бензина

    public TAuto(int Speed, int Massa, int Payload,
    int Petrol)
        : base(Speed, Massa, Payload)
    {
        this.Petrol = Petrol;
    }

    public void Print()
    {
        base.Print();
        Console.WriteLine("Petrol = {0}", Petrol);
    }
}

class TElectro : TTransport
{
    protected int Voltage; // Напряжение
```

```
        public TElectro(int Speed, int Massa, int Payload,
        int Voltage)
            : base(Speed, Massa, Payload)
        {
            this.Voltage = Voltage;
        }

        public void Print()
        {
            base.Print();
            Console.WriteLine("Voltage = {0}", Voltage);
        }
    }
```

Разберем, как работает наследование класса. Чтобы указать, что класс A является наследником класса B, следует написать:

```
class A : B
{
    ...
}
```

Теперь класс B имеет все поля и методы класса A. Однако поля и методы класса A, имеющие классификатор доступа private, будут недоступны для класса B. Если в классе B объявлены поля и методы, совпадающие с полями и методами класса B, то они будут заменены, или как говорят — сокрыты соответствующими полями и методами класса B. В нашем примере таким методом является метод Print. Если нам нужно из класса B обратиться к полям или методам родительского класса A, то для этого нужно использовать ключевое слово base, аналогичное ключевому слову this.

Если родительский класс имеет заданный конструктор, то он должен быть вызван из конструктора наследного класса. Обратите внимание, как можно вызывать конструктор родительского класса:

```
public TAuto(int Speed, int Massa, int Payload,
int Petrol) : base(Speed, Massa, Payload)
{ ... }
```

По практикуемся в создании классов наследников. Создадим два класса (автомобиль и такси) наследников от класса TAuto и два класса (трамвай и троллейбус) наследников от класса TElectro. У каждого нового класса могут быть дополнительные поля и методы, но мы для простоты не будем добавлять новых полей и методов.

```
class TBus : TAuto
{
    public TBus(int Speed, int Massa, int Payload,
    int Petrol) : base(Speed, Massa, Payload, Petrol) { }
}

class TTaxi : TAuto
{
    public TTaxi(int Speed, int Massa, int Payload,
    int Petrol) : base(Speed, Massa, Payload, Petrol) { }
}

class TTram : TElectro
{
    public TTram(int Speed, int Massa, int Payload,
    int Voltage) : base(Speed, Massa, Payload, Voltage) { }
}

class TTroll : TElectro
```

```
{
    public TTroll(int Speed, int Massa, int Payload,
    int Voltage) : base(Speed, Massa, Payload, Voltage) { }
}
```

Предположим, что у нас будет список различных транспортных средств, для которых нам нужно вычислить отношение стоимости единицы полезной массы к расходу топлива (бензина или электроэнергии). Создадим массив транспортных средств:

```
TTransport[] Trans;
Trans = new TTransport[4];
Trans[0] = new TAuto(120, 5, 2, 60);
Trans[1] = new TTaxi(250, 1, 1, 40);
Trans[2] = new TTram(100, 7, 5, 3000);
Trans[3] = new TTroll(80, 4, 2, 2500);
```

Заметим, что у нас в массиве Trans все элементы из разных классов, хотя и родственники класса TTransport. Конечно, мы могли бы, перебирая массив Trans, для каждого экземпляра класса подсчитать нужное соотношение. Но тут возникает проблема — ведь у половины классов нужно делить значение Payload на значение Petrol, а у половины нужно делить значение Payload на значение Voltage. Тем более, что идеология объектно-ориентированного программирования подразумевает, что классы сами должны сами обрабатывать собственные данные. Поэтому мы поступим следующим образом — мы добавим в наши классы метод Calc(). При этом мы воспользуемся технологией полиморфизма. **Полиморфизм** — это возможность объектов с одинаковой спецификацией иметь различную реализацию. Мы создадим у класса TTransport виртуальный метод:

```
class TTransport
{
```

```
        ...
        public virtual double Calc()
        {
            return 0;
        }
    }
```

Ключевое слово virtual означает, что наследники данного класса смогут изменить код данного метода. Сейчас мы еще модифицируем у родительского класса TTransport метод Print следующим образом:

```
        public void Print() // распечатать информацию
        {
            Console.WriteLine("Speed = {0}", Speed);
            Console.WriteLine("Massa = {0}", Massa);
            Console.WriteLine("Payload = {0}", Payload);
            Console.WriteLine("Calc Result = {0}", Calc());
        }
```

Теперь распечатаем наш массив следующим образом:

```
            foreach (TTransport T in Trans)
            {
                T.Print();
                Console.WriteLine();
            }
```

Мы увидим следующее:

```
Speed = 120 Massa = 5 Payload = 2 Calc Result = 0

Speed = 250 Massa = 1 Payload = 1 Calc Result = 0

Speed = 100 Massa = 7 Payload = 5 Calc Result = 0
```

```
Speed = 80 Massa = 4 Payload = 2 Calc Result = 0
```

Разумеется, везде мы видим «Calc Result = 0», потому что мы еще нигде не реализовали метод вычисления нужного нам коэффициента. Но вспомним, что мы пометили метод Calc как виртуальный, и теперь мы можем заменить этот метод у классов потомков. Сделаем следующие модификации наших классов:

```
class TAuto : TTransport
{
    ...

    public override double Calc()
    {
        return (double)Payload / (double)Petrol;
    }
}

class TElectro : TTransport
{
    ...

    public override double Calc()
    {
        return (double)Payload / (double)Voltage;
    }
}
```

Теперь наша программа выдаст нужный результат:

```
Speed = 120 Massa = 5 Payload = 2 Calc Result = 0,0333333333333333
```

```
Speed = 250 Massa = 1 Payload = 1 Calc Result = 0,025

Speed = 100 Massa = 7 Payload = 5 Calc Result = 0,001666666666667

Speed = 80 Massa = 4 Payload = 2 Calc Result = 0,0008
```

Чтобы оценить всю мощь полиморфизма нужно обратить внимание, что метод Calc вызывается в методе Print в методе TTransport! Ведь при наследовании мы можем и не иметь исходного текста класса TTransport, этот класс может быть доступен только в виде откомпилированной библиотеке. Тем не менее механизм полиморфизма позволяет нам модифицировать виртуальные методы. Мы будем использовать систематически полиморфизм при реализации вычислительных процедур.

Сделаем еще одно замечание. В родительском классе TTransport нам пришлось фиктивно реализовать метод Calc, хотя с точки зрения проектирования это не очень правильно. Во-первых, плохо писать код только для того, чтобы он был, а, во-вторых, плохо писать неверный код: возвращать 0. К счастью, развитое объектно-ориентированное программирование позволяет создавать так называемые абстрактные классы. Абстрактные классы могут содержать абстрактные виртуальные методы. Абстрактный метод содержит только объявление, но не имеет реализации. Разумеется, абстрактный метод является виртуальным, но ключевое слово virtual не пишется, чтобы в наследных классах можно было перекрыть этот метод. Более того, невозможно создать экземпляр абстрактного класса, а наследник абстрактного класса или сам должен быть абстрактным или обязан реализовать все абстрактные классы. Последний штрих к нашей программе:

```
abstract class TTransport
{
```

```
        ...

        public abstract double Calc();
    }
```

Еще раз подчеркнем, что мы не привели в этом листинге реализацию метода Calc не потому, чтобы сократить письмо, а потом что для абстрактного метода не бывает реализации.

Ключевые термины

Абстрактный класс — класс, содержащий нереализованные методы, которые необходимо реализовать в классах наследниках.

Наследование — технология проектирования классов, позволяющая создавать новые классы на базе предыдущих классов.

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Краткие итоги: На примерах проектирования классов продемонстрирована технологии наследования и полиморфизма в языке C# . Показано, как работать с абстрактными классами.

Глава II

Основные принципы вычислительной математики

4. Особенности вычислительных процедур

Вычисления сопровождают человечество с незапамятных времен. И уже в древности человечество стало использовать уравнения. Сейчас в математике используются огромное количество уравнений всех видов. Если посмотреть работы, например, по дифференциальным уравнениям, то можно заметить, что огромная часть работ посвящена доказательствам теорем существования, единственности, гладкости и качественным вопросам поведения решений. При этом вопрос о собственно нахождении решений часто вообще не возникает. А между прочим, уравнения создаются именно для нахождения решений. Ни сколько не умоляя важность теоретического исследования уравнений и свойств их решений, мы обратим внимание на нахождение решения.

Сразу возникает вопрос: а что значит решить уравнение? Что должно являться решением уравнения? Для простейшего линейного уравнения с *рациональными* коэффициентами — решение есть рациональное число, которое может быть представлено дробью с целыми числами и для нахождения которого необходимо выполнить конечное

число операций сложения и умножения целых чисел. А если вспомнить, что умножение есть многократное суммирование, то единственные операции, которые мы должны выполнять — это сложение целых чисел. Сложение целых чисел является **конструктивной операцией**. Сложить два целых числа может и ребенок и даже механическое устройство. И что самое главное — решение представимо в конечном виде.

Однако уже в древности люди столкнулись с тем, что некоторые очень простые, и взятые из практики, задачи нельзя решить в конечном виде. Примером тому простейшее квадратное уравнение:

$$x^2 - 2 = 0. \tag{II.1}$$

В тоже время любой (уже не любой, но хороший) школьник знает, что уравнение (II.2) имеет два решения:

$$x_{1,2} = \pm\sqrt{2}.$$

Однако еще Пифагор знал, что длина диагонали квадрата не соизмерима со стороной, т.е. число $\sqrt{2}$ не является рациональным и не может быть записана в конечном виде. Но так ли это? Что значить «быть выраженным в конечном виде»? Для нас это эквивалентно конструктивности объекта. Мы будем предполагать, что целые числа, операция сложения целых чисел, а также процедура сравнения двух целых чисел являются конструктивными. Далее конструктивный объект — это такой объект, который с любой наперед заданной точностью можно выразить с помощью конечного числа операций сложения (вычитания) целых чисел.

Мы подошли к центральному понятию вычислительных процедур — понятию **алгоритма**. Понятие алгоритма, как последовательность определенных действий, позволяющая за конечное время придти к заданному результату, является не формальным, поэтому в теории

алгоритмов используются так называемые уточнения понятия алгоритма. Среди наиболее известных уточнений алгоритма упомянем: частично рекурсивные функции, машины Тьюринга, нормальные алгорифмы Маркова, структурированные программы и др. Смысл уточнения понятия алгоритма состоит в том, чтобы выделить класс задач, которые могут быть решены с помощью идеальных вычислительных машин. Мы будем смотреть на понятие алгоритма с практической точки зрения. Все алгоритмы, которые мы будем рассматривать будут реализованы в виде программ на языке C# . Однако нужно помнить при реализации алгоритмов на языках программирования, что наличие в этих языках переменных типа double отнюдь не означает, что наши программы могут работать с вещественными числами, поскольку тип double может представлять только конечный набор рациональных чисел.

Возвращаясь к числу $\sqrt{2}$, покажем как записать это число в конечном виде. Для этого приведем код на языке C# , который вычисляет значение $\sqrt{2}$ с любой наперед заданной точностью.

```
double a = 1;
double b = 2;
double c;

// prec = нужная точность

while ((b - a) > prec)
{
    c = a + (b - a) / 2.0;

    if (f(c) > 0)
    {
        b = c;
```

```
            }
            else
            {
                a = c;
            }
        }
// теперь в переменной c нужное значение
```

В математике долгое время найти решение означало выписать «аналитически» решение: число или функцию. Однако с возрастающими потребностями инженерных вычислений все большее значения стали приобретать расчеты «вплоть до числа». В этой ситуации, во-первых, для большинства задач не существовало «аналитически выражаемого» решения, а, во-вторых, зачастую нахождение аналитического решения и последующее вычисление конкретных чисел представляло собой не более легкую задачу, чем вычисления приближенного решения.

Однако, если не брать во внимание мифическую разностную машину Ч.Бебиджа, то вычисления производились в ручную (с использованием подручных логарифмических линеек и арифмометров). Следовательно, численные методы разрабатывались как можно более удобно для ручных расчетов. Появление программируемых вычислительных машин кардинально изменило отношение к вычислительным процедурам. С одной стороны открылись огромные возможности численных экспериментов, а с другой перед вычислительной математикой возникли новые проблемы, связанные с машинной точностью.

Проведем небольшой вычислительный эксперимент, который продемонстрирует существование машинного ε. **Машинным** ε называется такое положительное число ε, для которого на рассматриваемой

машине является верным утверждение:

$$1 = 1 + \varepsilon.$$

Разумеется, с точки зрения математики существование машинного ε является невозможным. Но дело в том, что при реализации на компьютере банальные арифметические операции не совпадают с настоящими арифметическими операциями. Это связано с конечностью представления чисел на компьютере. Приведем код для вычисления машинного ε.

```
class TEps {
    public double CalcEps()
    {
        double eps = 1;

        while (1.0 != 1.0 + eps)
        {
            eps = eps / 10.0;
            Console.WriteLine(eps);
        }

        return eps;
    }
}
```

На моем компьютере машинное $\varepsilon \leq 10^{-16}$. Наличие машинного ε не означает, что невозможно использовать числа меньшие по модулю. Согласно документации .NET число с типом double может быть порядка от $\pm 5 \cdot 10^{-324}$ до $\pm 1.7 \cdot 10^{308}$. Но при выполнении машинных расчетов нужно учитывать машинную арифметику.

Чтобы продемонстрировать отличие машиной арифметики рас-

смотрим следующий пример. Вычислим две величины:

$$A = \sum_{k=1}^{100} \frac{1}{10^k}$$

и

$$a = \sum_{k=100}^{1} \frac{1}{10^k}$$

Разница между этими величинами только в порядке суммирования. Согласно школьному переместительному закону эти величины равный! Но так ли это в мире программировании? Проверим:

```
class TEps {
    double[] ak;

    public TEps()
    {
        ak = new double[101];
        ak[0] = 1.0;
        for (int k = 1; k <= 100; k++)
        {
            ak[k] = ak[k - 1] / 1.1;
        }
    }

    public double A()
    {
        double res = 0;

        for (int k = 1; k <= 100; k++)
        {
            res += ak[k];
        }
```

```
        return res;
    }

    public double a()
    {
        double res = 0;

        for (int k = 100; k >= 1; k--)
        {
            res += ak[k];
        }

        return res;
    }
}
```

Если мы вычислим разницу между величинами A и a, то получим:

$$A - a = 3,5527136788005 \cdot 10^{-15}.$$

При вычислении сумм всегда нужно суммировать с наименьших по модулю чисел (почему?).

Следующая важная особенность вычислительных процедур — это **вычислительная неустойчивость**. Как мы уже видели, помимо погрешностей численных методов в расчетах всегда возникают ошибки округления. Эти ошибки могут накапливаться в ходе расчетов и сделать результат вычислений совершенно неверным. Вычислительная неустойчивость также является препятствием для повышения точности расчетов. Продемонстрируем это на примере вычисления производной функции $f(x) = x^2$. Мы будем вычислять эту функцию в разных точках с различным приращением аргумента по естествен-

ной формуле:

$$f'(x_0) = \frac{f(x+h) - f(x)}{h}. \qquad \text{(II.2)}$$

Функция $f(x) = x^2$ является более чем дифференцируемой в любой точке, поэтому (теоретически) чем меньше приращение h тем с большой точностью мы должны приближаться к значению производной: $f'(x_0) = 2x_0$.

А теперь посмотрим как это выглядит на практике. Реализуем эту процедуру:

```
public double Dx2(double x0, double h) {
    double res = 0;

    double a, b;

    a = x2(x0 + h);
    b = x2(x0);

    res = (a - b) / h;

    return res;
}

double x2(double x) {
    return x * x;
}
```

Опробуем эту процедуру наивно, выбрав очень маленькое приращение:

```
Console.WriteLine("x0 = 10, h = 1e-20 Dx2 = {0}", Eps.Dx2(10,
1e-20));
```

Получаем:

```
x0 = 10, h = 1e-20 Dx2 = 0
```

Абсурдный результат! Но здесь все просто, мы выбрали приращение заранее меньшее чем машинное ε, поэтому в реальности нашего приращения попросту не было. Теперь будем осторожнее:

```
x0 = 1000000, h = 1e-9 Dx2 = 2075195,3125
```

Конечно, это уже что-то похоже на правду, но хотелось бы улучшить точность. Для этого уменьшим приращение, но так, чтобы наше приращение было заведомо меньше машинного ε:

```
x0 = 1000000, h = 1e-12 Dx2 = 0
```

Результат обескураживающий! По теории точность должна увеличиться, а вместо этого результат нулевой. А попробуем теперь *увеличить* приращение:

```
x0 = 1000000, h = 1e-6 Dx2 = 2000000
```

Мы получили верный результат, хотя по теории точность должна уменьшаться. Этот простой пример показывает, что при программировании вычислительных процедур необходимо учитывать, что машинная арифметика отличается от обычной арифметики.

Теперь рассмотрим вопрос об эффективности вычислительных процедур. Математика не знает оценочных категорий, такие слова «много», «сложно», «долго» не имеют математического смысла. Однако когда мы начинаем реализацию вычислительных процедур на реальных компьютерах, а не на машине Тьюринга, то вопросы эффективности вычислительных процедуры могут стать принципиальными. Например, с точки зрения «чистой математики» разложить любое целое число на простые сомножители не представляет собой никаких проблем — любое число может быть единственным образом разложено на простые сомножители, и можно указать алгоритм

для этого разложения. Однако это знание не позволит выполнить такое разложение для больших чисел. Другой пример представляют собой задачи целочисленного линейного программирования, в которых необходимо найти минимум линейной функции на заданном компактном множестве в $\mathbb{R}^n$ с целочисленными координатами. Поскольку компактное множество конечно, то количество точек, на которых необходимо найти минимум, конечно. Следовательно, можно утверждать существования такого минимума и с помощью перебора найти точки, на которых этот минимум достигается. Однако для решения этой задачи разрабатывают многочисленные методы решения задачи линейного программирования, пригодные для проведения реальных расчетов. В дискретной математике изучается большое количество «конечных» задач, которые теоретическим имеют тривиальное решение путем перебора. Однако реальность требует разрабатывать оптимальные методы для решения этих задач.

Ключевые термины

Алгоритм — последовательность определенных действий, позволяющая за конечное время придти к заданному результату.

Вычислительная неустойчивость — неустойчивость при работе вычислительной процедуру в следствии машинной арифметики.

Конструктивная операция — операция, которая может быть выполнена механически в конечное время и при использовании конечного алфавита.

Машинное ε — такое положительное число ε, для которого на рассматриваемой машине является верным утверждение $1 = 1 + \varepsilon$.

Краткие итоги: На простых, но естественных примерах показаны особенности вычислительных процедур при их реализации на компьютере. Показаны простейшие методы разработки вычислительных процедур, позволяющие избежать вычислительной неустойчивости.

5. Принципы организации вычислительных процедур

К классическим темам вычислительной математики обычно относят следующие разделы:

- Табулирование функций и интерполяция
- Нахождения корней систем уравнений
- Решение задач линейной алгебры
- Численное интегрирование и дифференцирование
- Решение обыкновенных дифференциальных уравнений (задачи Коши и краевых задач)
- Решение уравнений в частных производных
- Решение интегральных уравнений
- Задачи линейного и нелинейного программирования
- Обработка результатов экспериментов и задачи математической статистики
- Решение задач дискретной математики

Разумеется, это не полный список тем вычислительной математики, поскольку практически в любой области математики можно найти «работу» для численных методов. Среди собственных тем вычислительной математики можно выделить следующие:

- Сходимость численных методов
- Оценка погрешности и сложности вычислительных процедур
- Программирование численных методов

- Методика проведения вычислительных экспериментов

Рассмотрим принципы организации вычислительных процедур на примере решения **операторного уравнения**. Пусть X и Y метрические пространства с метриками ρ_X и ρ_Y соответственно, и пусть задано однозначное отображение

$$A : X \to Y.$$

Рассматривается уравнение

$$A[x] = y, \tag{II.3}$$

где $y \in Y$.

Решением уравнения (II.3) называется элемент $x \in X$, удовлетворяющий уравнению (II.3). Мы будем предполагать, что существует хотя бы одно решение уравнения (II.3), которое мы обозначим через x^*. Задачей вычислительной математики является конструктивное построение такого отображения

$$T : \mathbb{N} \to X,$$

что последовательность

$$x_n = T[n]$$

удовлетворяет следующему условию:

$$\rho_X(x_n, x^*) \to 0, \quad n \to \infty.$$

Для оценки эффективности вычислительной процедуры введем две величины: оценка погрешности приближенного решения

$$\delta(x) = \rho_X(x, x^*),$$

и **невязка приближенного решения**:

$$\varepsilon(x) = \rho_Y(A[x], y).$$

Для вычисления величины $\delta(x)$ необходимо знать собственно решение x^*, что возможно лишь в тестовых задачах. С другой стороны при вычислении невязки решение не участвует, поэтому величина $\varepsilon(x)$ является конструктивно вычисляемой.

Как правило, целью численных методов является построение такой последовательности x_n, для которой выполнено условие:

$$\delta(x_n) \to 0.$$

Если же для нашей последовательности выполнено только условие

$$\varepsilon(x_n) \to 0,$$

то это еще не означает, что последовательность x_n сходиться к решению (и вообще, что сходится). Рассмотрим простой пример. Пусть $X = Y = \mathbb{R}$ — числовая ось. Функция A задана по формуле:

$$A[x] = \frac{x^2}{1 + x^4}.$$

Рассмотрим уравнение

$$\frac{x^2}{1 + x^4} = 0.$$

Это уравнение имеет единственное решение $x^* = 0$. Если мы используем вычислительную процедуру

$$x_n = T[n] = \frac{1}{n},$$

то легко видеть, что невязка $\varepsilon(x_n)$ имеет вид

$$\varepsilon(x_n) = \frac{n^2}{1 + n^4} \to 0, \quad n \to \infty.$$

Однако последовательность x_n никак не сходится к решению. Заметим, что причиной такой ситуации является то, что пространство X, на котором задана функция A, не является компактным.

Однако есть следующая зависимость между функциями δ и ε. Если предположить, что отображение A является непрерывным отображением, то для любой последовательности x_n из условия

$$\delta(x_n) \to 0.$$

следует

$$\varepsilon(x_n) \to 0.$$

Обратное, как мы только что видели, не верно.

Обратимся к способам построения операторов $T[n]$. Мы рассмотрим два принципиальным методам построения приближенных решений. Во-первых, это метод дискретизации. Как правило, пространство X состоит из бесконечного числа элементов, более того, часто это пространство является бесконечномерным линейным пространством. А на компьютере мы можем работать только с конечными множествами. Поэтому для каждого целого n вводится конечное множество X_n и два отображения:

$$D_n : X \to X_n$$

и

$$I_n : X_n \to X.$$

Первое отображение осуществляет дискретизацию бесконечного пространства. Примером этого отображения может быть сужение непрерывной функции, заданной на отрезке на конечную сетку. Второе отображение играет роль интерполяции с конечного множества. Часто для этих отображений выполнено условие согласования:

$$D_n I_n \tilde{x} = \tilde{x},$$

для любого элемента $\tilde{x} \in X_n$.

Далее решается следующая задача

$$A[I_n(x)] = y. \tag{II.4}$$

Для этой задачи необходимо выбрать элемент $\tilde{x}_n \in X_n$, тогда оператор T строится по формуле:

$$T[n] = I_n \tilde{x}_n.$$

Поскольку выбор элемента $\tilde{x}_n$ осуществляется из конечного множества, то такой выбор может быть осуществлен для заданного n на вычислительной машине.

Вычислительные процедуры, основанные на методе дискретизации пространства, как правило, применяются для нахождения приближенных решений дифференциальных и интегральных уравнений. В том числе и для приближенного решения уравнений в частных производных.

Другим часто применяемым принципом построения вычислительных процедур является использование **принципа неподвижной точки отображения**. Предположим, что решение уравнения (II.3) является неподвижной точкой некоторого другого отображения $B : X \to X$, то есть если x^* — решение задачи (II.3), то имеет место:

$$Bx^* = x^*. \tag{II.5}$$

Для простоты рассмотрения мы будем предполагать, что отображение B имеет единственную неподвижную точку.

Во многих случаях для конструктивного нахождения неподвижной точки оператора B или, соответственно, решения уравнения (II.5) используется метод простых итераций. Пусть задано начальное приближение $x_0 \in X$, тогда последующие приближения строятся по формуле:

$$x_n = Bx_{n-1}, \quad n = 1, 2, \ldots$$

В случае, когда оператор B является сжимающим отображением, то оператор B имеет единственную неподвижную точку и последовательность x_n сходится к этой точке при любом выборе начального приближения.

Напомним, что отображение B называется сжимающим, если существует такое число $\alpha < 1$, что для любых $x', x'' \in X$ имеет место

$$\rho(Bx', Bx'') \leq \alpha\rho(x', x'').$$

Рассмотрим пример на применение метода неподвижных точек. Пусть пространства X и Y являются пространствами $\mathbb{R}^2$ с евклидовой метрикой. В качестве оператора A возьмем квадратную матрицу:

$$A = \begin{pmatrix} 0.9 & -0.4 \\ -0.2 & 0.8 \end{pmatrix}$$

и рассмотрим уравнение

$$Ax = f, \tag{II.6}$$

где $f = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ — вектор столбец. Таким образом, уравнение (II.6) представляет собой систему линейных алгебраических уравнений. Чтобы использовать метод неподвижной точки введем оператор B следующим образом:

$$Bx = \begin{pmatrix} 0.1 & 0.4 \\ 0.2 & 0.2 \end{pmatrix} x + \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Легко видеть, неподвижная точка оператора B совпадает с точным решением уравнения (II.6). Кстати, точное решение этого уравнения имеет вид:

$$x^* = \begin{pmatrix} 1.875 \\ 1.71875 \end{pmatrix}$$

Будем строить приближенные решения по формуле

$$x_n = Bx_{n-1}, \quad n = 1, 2, \ldots$$

где $x_0 = 0$. Знание точного решения позволит нам вычислять значения $\delta(x_n)$. Приведем программу расчета.

```
// Класс - двумерный вектор
class TR2
```

```
{
    public double x1; // координаты
    public double x2;

    // конструктор по умолчанию
    public TR2()
    {
        x1 = 0;
        x2 = 0;
    }

    // конструктор с заданными значениями
    public TR2(double x1, double x2)
    {
        this.x1 = x1;
        this.x2 = x2;
    }

    // вычислить метрику до точки A
    public double rho(TR2 A)
    {
        return Math.Sqrt((x1 - A.x1) * (x1 - A.x1) +
        (x2 - A.x2) * (x2 - A.x2));
    }

}

// Класс - матрица 2x2
class TMatrix2
{
    double a11, a12, a21, a22;
```

```
    public TMatrix2(double a11, double a12,
        double a21, double a22)
    {
        this.a11 = a11;
        this.a12 = a12;
        this.a21 = a21;
        this.a22 = a22;
    }

    // вычислить вектор после применения матрицы к вектору x
    public TR2 Calc(TR2 x)
    {
        TR2 res = new TR2();

        res.x1 = a11 * x.x1 + a12 * x.x2;
        res.x2 = a21 * x.x1 + a22 * x.x2;

        return res;
    }
}

// Класс для итерационного решения уравнения
class TContractingMatrix2
{
    TR2 x, f; // текущее значение, правая часть
    TMatrix2 Matrix; // матрица оператора

    public TContractingMatrix2(TMatrix2 Matrix, TR2 x0, TR2 f)
    {
        this.Matrix = Matrix;
```

```
            x = x0;
            this.f = f;
        }

        // переход к следующей итерации
        public TR2 Next()
        {
            TR2 xx = Matrix.Calc(x);

            xx.x1 += f.x1;
            xx.x2 += f.x2;

            x = xx;

            return x;
        }

        // текущее состояние
        public TR2 GetCurent()
        {
            return x;
        }
    }
```

Теперь испытаем класс TContractingMatrix2 на нашем примере.

```
x_n=[1.000000E+000,1.000000E+000]; d_n=1.132354E+000
x_n=[1.500000E+000,1.400000E+000]; d_n=4.921652E-001
x_n=[1.710000E+000,1.580000E+000]; d_n=2.155842E-001
x_n=[1.803000E+000,1.658000E+000]; d_n=9.420490E-002
x_n=[1.843500E+000,1.692200E+000]; d_n=4.119651E-002
x_n=[1.861230E+000,1.707140E+000]; d_n=1.801125E-002
```

```
x_n=[1.868979E+000,1.713674E+000]; d_n=7.875165E-003
x_n=[1.872368E+000,1.716531E+000]; d_n=3.443224E-003
x_n=[1.873849E+000,1.717780E+000]; d_n=1.505477E-003
x_n=[1.874497E+000,1.718326E+000]; d_n=6.582366E-004
x_n=[1.874780E+000,1.718564E+000]; d_n=2.877996E-004
x_n=[1.874904E+000,1.718669E+000]; d_n=1.258341E-004
x_n=[1.874958E+000,1.718715E+000]; d_n=5.501820E-005
x_n=[1.874982E+000,1.718734E+000]; d_n=2.405550E-005
x_n=[1.874992E+000,1.718743E+000]; d_n=1.051774E-005
x_n=[1.874996E+000,1.718747E+000]; d_n=4.598653E-006
x_n=[1.874998E+000,1.718749E+000]; d_n=2.010661E-006
x_n=[1.874999E+000,1.718749E+000]; d_n=8.791174E-007
x_n=[1.875000E+000,1.718750E+000]; d_n=3.843748E-007
x_n=[1.875000E+000,1.718750E+000]; d_n=1.680595E-007
x_n=[1.875000E+000,1.718750E+000]; d_n=7.348034E-008
x_n=[1.875000E+000,1.718750E+000]; d_n=3.212767E-008
x_n=[1.875000E+000,1.718750E+000]; d_n=1.404712E-008
x_n=[1.875000E+000,1.718750E+000]; d_n=6.141797E-009
x_n=[1.875000E+000,1.718750E+000]; d_n=2.685366E-009
```

Мы видим, что величина $\delta(x_n)$ стремится к нулю, а сама последовательность x_n сходится к точному решению.

Классический подход вычислительной математики состоит в том, что строится некоторая эффективно выполняемая вычислительная процедура позволяющая получать приближенные решения, которые сходятся (в том или ином смысле) к точному решению. Такой подход подразумевает, что исходная задача является корректной. **Корректность задачи по Адамару** означает выполнения следующих трех условий:

- Существование решения
- Единственность решения

- Непрерывная зависимость решения от оператора задач, правой части, начального условия

В начале прошлого века могло показаться, что эти условия всегда должны быть выполнены для имеющих физический смысл задач. Однако довольно быстро выяснилось, что наука и техника предъявляют большой класс задач, для которых условия некоторые из условий 1–3 не выполнены. К таким задачам относятся уравнения Фредгольма первого рода, обратные задачи для дифференциальных уравнений, задачи аналитического продолжения функций и многие другие. Задачи, для которых не выполнены какие-либо из условий 1–3, называются некорректными задачами. Некорректные задачи возникают в физике, например, неустойчивость Релея-Тейлора.

Как правило, применять классические вычислительные процедуры для решения некорректных задач не удается. В середине XX-го века известным математиком А.Н. Тихоновым были заложены основы теории регуляризации некорректных задач. Эти результаты нашли свое эффективное применение во многих научно-технических областях.

Изучение теории регуляризации некорректных задач выходит за рамки нашего курса.

Ключевые термины

Корректность задачи по Адамару — существование единственного решения у задачи и непрерывная зависимость решения от данных задачи.

Невязка приближенного решения — количественная мера неудовлетворения приближенным решением уравнению.

Операторное уравнение — уравнение в абстрактных пространства записанное с помощью операторов в этих пространствах.

Принцип неподвижной точки отображение — существование такого элемента для оператора, который отображается этим опе-

ратором в себя.

Краткие итоги: Рассмотрены различные способы организации вычислительных процедур для решения операторных уравнений. Рассмотрены вопросы связи невязки и точности решений. Приведен пример организации итерационной процедуры.

6. Научное программирование

Изначально программирование предназначалось для решения именно научных и инженерных задач. Но с развитием вычислительной техники задачи, которые решались с помощью компьютеров, стали значительно шире. В настоящее время офисное программирование, программирование компьютерных игр, поисковые системы, компьютерная графика, базы данных — далеко не полный спектр решаемых на компьютерах задач. Программирование этих и многих других задач существенно отличается от программирование научных расчетов. Поэтому имеет смысл выделять научно программирование, как отдельный стиль программирования.

Что значит **научное программирование**? Прежде всего научное программирование должно быть ориентировано на эффективное и корректное решение поставленной задачи. Можно возразить, что эти требования предъявляются ко всем программным средствам. Однако вопрос в приоритете критериев. Скажем, если офисная программа будет работать не $0,25$ секунд, а 1 секунду для обработки операции, то это не так критично, как скажем проведение расчета в течении недели или месяца. Тоже самое и на счет корректности выполнения программы. Одно дело, когда «зависнет» компьютерная игра, и совсем другое дело, если в ходе вычислений будет ошибка, которая приведет к неверным выводам или технологическим катастрофам.

Приведем некоторые моменты, которые следует иметь в виду,

приступая к программированию научных задач.

Отделение научной части от интерфейсной. Мы призываем придерживаться правила: одна программа производит численный расчет, сохраняя результаты расчета, а другая программа осуществляет визуализацию полученных данных.

Оптимизация текстов программ. Очень часто небольшое изменение в тексте программы позволяет кардинально увеличить скорость расчетов. Оптимизирующий компилятор не всесилен.

Не использовать внешние подпрограммы и библиотеки. За исключением особых случаев старайтесь программировать используемые алгоритмы сами. Во-первых, это даст исчерпывающее понимание самого численного метода, а, во-вторых, реализуя «для себя» можно добиться наилучшего результата.

Жертвуйте универсальностью в угоду эффективности. Универсальность одно из самых любимых достижений программирования, но для научного программирования построение универсальных программных комплексов оправдано не всегда.

Используйте современные компиляторы. Далеко не все можно оптимизировать «руками» — современные оптимизирующие компиляторы могут серьезно увеличить скорость.

Обсудим вопрос выбора языка программирования. Мы в нашем курсе систематически используем язык C# . Издавна языком для научных расчетов являлся Фортран. Действительно, этот язык имеет много преимуществ, но еще больше недостатков. Таких как:

Язык старого поколения. Этот язык был разработан на заре компьютерной эры, поэтому многие важные технологии программирования такие как модульность, контроль типов и др. в нем

не реализованы. Что приводит к ошибкам и трудностям при программировании.

Многие библиотеки подпрограмм уже устарели. Прогресс в компьютерной сфере происходит крайне быстро. Поэтому библиотеки разработаны для устаревших ЭВМ могут быть бесполезны.

Отсутствие мобильности. Не смотря на существующие стандарты языка, конкретные реализации этого языка существенно отличаются.

Выбор языка C# мотивируется следующими факторами:

Современный язык. Программирование это та область, в которой прогресс происходит очень быстро. Язык C# с одной стороны вобрал в себя лучшие черты языков C++ и Java, а с другой был создан без необходимости поддерживать совместимость с предыдущими версиями

Мощность платформы .NET. Язык C# является основным языком платформы .NET, поэтому при программировании на C# вы программируете также и на платформе .NET, которая предоставляет вам мощные, хорошо продуманные классы.

Эффективность языка. Как ни странно, но программы, написанные на C# , выполняемые на .NET, являются очень эффективно выполняемыми приложениями. Однако при разработки сложных программ следует учитывать не только скорость выполнения программы, но и скорость разработки программы, а также сложно дальнейшей модификации. По этому параметру язык C# значительно превосходит классический язык C++.

В последние годы наблюдается возникновение новой вычислительной парадигмы - **облачных вычислений** (cloud computing),

при которой создатель программы полностью абстрагируется от того, где будет происходить вычислительный процесс. Облачные вычисления подразумевают наличие значительных вычислительных мощностей, доступных через Интернет по требованию. На этих мощностях, располагающихся в центрах обработки данных (ЦОД), возможно развертывание интернет-приложений, веб-сервисов, услуг хранения и обработки данных и т.д. За счет концентрации вычислительных мощностей в экономически-привлекательных районах, повсеместного использования виртуализации и автоматизации контроля за работоспособностью узлов поставщики облачных услуг могут добиться существенное экономии от масштаба и высокой доступности и надежности предоставляемых сервисов.

Облачное приложение состоит из набора сервисов, работающих в центре обработки данных. При этом все вопросы запуска сервисов и поддержания их работоспособности берет на себя среда поддержки вычислений - от программиста требуется реализовать соответствующий программный код и файл с описанием и произвести развертывание. Как правило, облачное приложение состоит из вычислительных компонентов (это может быть веб-сервис, веб-приложение или вычислительный блок, не требующий взаимодействия с пользователей) и хранилища данных, которое мы отличаем здесь от термина «база данных» , поскольку из-за необходимости масштабирования как правило применяются нереляционные хранилища. Вычислительные компоненты как правило программируются в модели «без состояния» (stateless), что дает возможность в случае отказа перезапустить соответствующий компонент без нарушения работоспособности приложения. Кроме того, среда позволяет запустить вычислительную компоненту на нескольких виртуальных серверах одновременно и использовать балансировку нагрузки для повышения производительности.

Следует отметить, что не любая задача может эффективно вычисляться на облачном кластере в такой архитектуре. В отличие от

традиционного компьютерного кластера, в котором имеется высокоскоростной канал обмена данными между узлами, в указанной архитектуре обмен осуществляется через низкоскоростное внешнее хранилище данных. Теоретически, возможно организовать прямой сетевой обмен между экземплярами вычислительных ролей, однако такой подход выходит за рамки настоящего курса. В любом случае следует понимать, что облачные вычисления не являются заменой высокопроизводительным вычислительным кластерам, и скорость обмена данными между узлами у них ниже. Тем не менее, есть значительное количество задач, которые не требуют существенного обмена данными между узлами в процессе вычислений. В качестве примера, можно рассмотреть задачи независимого перебора, в которых можно изначально разбить пространство поиска на непересекающиеся подпространства, поиск в которых будет производиться независимыми вычислительными ролями.

Ключевые термины

Научное программирование — стиль программирования, ориентированный на научные расчеты.

Облачные вычисления — вычисления, спроектированные таким образом, чтобы абстрагироваться от места проведения расчетов.

Краткие итоги: Рассмотрены черты научного программирования, даны рекомендации по повышению эффективности программ. Рассмотрены вопросы организации облачных вычислений.

Глава III

Объектно-ориентированная реализация числовых функций

7. Объектно-ориентированнная реализация действительных чисел

Основным понятием вычислительной математики и математики вообще является понятие числа. Способность к счету является фундаментальным свойством человека. Проблема, впрочем, состоит в том, что большинство разделов математика оперирует понятие действительного числа. Однако мы исходим из того, что механически можно выполнять лишь операции с целыми числами ($\mathbb{N}$). Под операциями мы понимаем операцию сложения, умножения на -1, а также сравнения двух целых чисел. Поскольку рациональное число представляет собой пару целых чисел (числитель/знаменатель), то эти же операции можно считать механическими и для рациональных чисел ($\mathbb{Q}$). Драма человечества состояла в открытии несоизмеримости некоторых величин. С этого момента нам потребовались числа, которые не являются механически обозримыми. С древности были известны некоторые иррациональные числа, такие как $\sqrt{2}$, π, e. Полноценная

теория действительных чисел требует применения методов математического анализа. Наиболее ясное построение действительных чисел основано на пополнении метрического пространства рациональных чисел. Идея этого пополнения состоит в том, что все фундаментальные последовательности рациональных чисел разбиваются на классы эквивалентности. Две фундаментальные последовательности рациональных чисел $\{a_n\}$ и $\{b_n\}$ объявляются эквивалентными если для любого $\varepsilon > 0$ существуют такое $N = N(\varepsilon)$, что для всех $p, q > N$ выполнено неравенство

$$|a_p - b_q| < \varepsilon.$$

Теперь с каждым классом эквивалентности фундаментальных последовательностей рациональных чисел связывается некоторый идеальный элемент, называемым действительным (или вещественным) числом.

Прежде чем продолжить рассмотрение действительных чисел, введем формально важнейшее понятие — понятие алгоритма. Интуитивно алгоритм понимается, как однозначное предписание элементарных действий. Согласно **тезису Черча-Тьюринга** любая интуитивно вычислимая функция может быть вычислена с помощью машины Тьюринга. Есть и другие эквивалентные формальные определения алогоритма на основе нормальных алгорифмов Маркова, частично вычислимых функций, структурных программ и др. Машина Тьюринга эквивалентна более простой **машине Поста**.

Гипотетическая машина Поста состоит из бесконечной ленты, разбитой на отельные ячейки. Каждая ячейка может быть помечена либо 0, либо 1. Также у машины Поста есть воображаемая каретка, которая в каждый момент может находиться только над одной из ячеек. Каретка управляется согласно программе, которая состоит из шести команд:

- $\longrightarrow$ сдвиг каретки вправо

- $\longleftarrow$ сдвиг каретки влево
- 0 записать в текущую ячейку 0
- 1 записать в текущую ячейку 1
- $?(I_1, I_2)$ если в текущей ячейке записано 1, то перейти к I_1-ой строке программы, иначе к строке I_0
- *Stop* остановка

Работа машины Поста задается программой, содержащей конечное число команд, и начальной конфигурацией (состояние ленты и положение каретки). Начальная конфигурация должна содержать лишь конечное число ячеек, содержащих 1. Мы будем говорить, что функция $f(n)$ натурального аргумента и принимающая натуральное значение, алгоритмически вычислима или просто вычислима для аргумента n, если существует какая машина Поста, что если начальная конфигурация содержит ровно n ячеек, содержащих 1, то работа машина обязательно закончится командой *Stop* и в момент остановки на ленте будет ровно $f(n)$ ячеек, содержащих 1.

Программирование даже простейших функций на машине Поста очень громоздко и сложно, поэтому мы будем обращаться к машине Поста лишь как к формальному определению алгоритма. Заметим еще, что поскольку можно построит алгоритм (и соответственно машину Поста), которая однозначно сопоставляет каждому натуральному числу — рациональное число, то мы будем рассматривать алгоритмически вычислимые функции и рациональных аргументов, и принимающие рациональные значения. И даже функции отображающую любое счетное множество в другое счетное множество.

Мы часто будем говорить, что **последовательность рациональных чисел a_n является вычислимой**, если существует такая вы-

числимая функция $f : \mathbb{N} \to \mathbb{Q}$, что

$$a_n = f(n).$$

Вернемся к вещественным числам. Данное выше определение (и любое другое) вызывает ряд вопросов. В самом деле, любое вещественное число — это класс эквивалентности фундаментальных последовательностей. Во-первых, для любого ли вещественного числа можно построить вычислимую фундаментальную последовательность? Ответ, очевидно, нет! Почему? Потому что множество вещественных чисел имеет мощность континуума, а количество всех алгоритмов (машин Поста) счетно. Во-вторых, даже если у нас есть вычислимая фундаментальная последовательность, то это еще не означает, что мы можем узнать предел этой последовательности.

Введем понятие вычислимой сходимости последовательности. Последовательность a_n называется вычислимо сходящейся к числу α, если существует такая вычислимая функция $\xi : \mathbb{Q} \to \mathbb{N}$, что для любого рационального $\varepsilon > 0$ выполнено

$$|\alpha - a_n| < \varepsilon,$$

для любого $n > \xi(\varepsilon)$.

Число α называется **конструктивным действительным числом**, если существует вычислимая последовательность рациональных чисел, которая вычислимо сходиться к α. Можно сказать, что конструктивное действительное число это пара вычислимых функций $\{a(n), \xi(\varepsilon)\}$. Можно конструктивное действительное число представить в виде одной вычислимой функции: $A : \mathbb{Q} \to \mathbb{Q}$, которая по заданному $\varepsilon > 0$ вычисляет рациональное число такое, что:

$$|\alpha - A(\varepsilon)| < \varepsilon.$$

Именно в таком виде мы и реализуем класс для конструктивного действительного числа.

Реализуем абстрактный класс, который будет заготовкой для классов различных конструктивных действительных чисел.

```
abstract class TCR
{
    public TCR()
    {
    }

    public abstract double A(double prec);
}
```

Разумеется, на компьютере мы можем реализовать лишь некоторое приближение к конструктивному действительному числу. Действительно, переменная *prec*, во-первых, не может быть произвольным рациональным числом, а, во-вторых, не с любой точностью мы можем проводить вычисления. И еще заметим, что переменные типа *double* представляют собой некоторое подмножество рациональных чисел.

Возникает естественный вопрос — а какие есть примеры конструктивных действительных чисел, не являющихся рациональными? Конструктивными действительными числами являются, например, $\sqrt{2}$, π, e, $\sin(1)$. Для примера реализуем первые два из этих конструктивных чисел. Для вычисления числа π мы воспользуемся рядом Лейбница:

$$\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Конечно, есть ряды, которые значительно быстрее сходятся к числу π, но мы выбрали ряд Лейбница по причине, что для знакопеременного ряда остаток ряда оценивается через модуль последнего слагаемого частичной суммы.

```
class TCPi : TCR
{
    public override double A(double prec)
    {
        int n = 1;
        double res = 0;
        double an = -1.0;

        double l = 1.0;

        do
        {
            an = 4.0 / (double)n * l;
            res += an;
            l = -l;
            n += 2;
        }
        while (Math.Abs(an) > prec);

        return res;
    }
}
```

Для построение конструктивного действительного числа $\sqrt{2}$ мы реализуем метод дихотомии для решения уравнения

$$x^2 - 2 = 0, \quad x \in [1, 2].$$

Вот этот класс.

```
class TCSQRT2 : TCR
{
    public override double A(double prec)
```

```
    {
        double a = 1;
        double b = 2;
        double c;

        while ((b - a) > prec)
        {
            c = a + (b - a) / 2.0;

            if (f(c) > 0)
            {
                b = c;
            }
            else
            {
                a = c;
            }
        }

        return a;
    }

    double f(double x)
    {
        return x * x - 2.0;
    }
}
```

Возникает еще один вопрос, если среди действительных чисел не конструктивные действительные числа, то значит есть (и их континуум!) неконструктивных действительных чисел. А есть ли пример

такого числа? Но на бумаге невозможно описать неконструктивный объект — можно лишь доказать (неконструктивно) его существование.

Заметим, что в рамках конструктивной математики работать с конструктивными действительными числами весьма сложно. Например доказывается теорема о том, что для произвольного конструктивного действительного числа нет алгоритма, позволяющего определить равно это число нулю или нет. В частности, в рамках конструктивных действительных чисел простейшее линейное уравнение

$$ax = b$$

неразрешимо.

Однако есть операции для конструктивных действительных чисел, которые являются алгоритмически разрешимыми, например сложение двух конструктивных чисел. Реализуем соответствующий класс

```
class TCRAdd : TCR
{
    TCR CR1, CR2;

    public TCRAdd(TCR CR1, TCR CR2) : base()
    {
        this.CR1 = CR1;
        this.CR2 = CR2;
    }

    public override double A(double prec)
    {
        return CR1.A(prec / 2.0) + CR2.A(prec / 2.0);
    }
}
```

Ключевые термины

Вычислимая последовательность чисел — числовая последовательность, каждый элемент которой может быть получен с помощью вычислимой функции по номеру.

Конструктивное действительное число — число, являющееся вычислимым пределом вычислимой последовательности рациональных чисел.

Машина Поста — гипотетическая вычислительная машина, эквивалентная машине Тьюринга.

Тезис Черча-Тьюринга — утверждение, что любая интуитивно вычислимая функция может быть вычислена с помощью машины Тьюринга.

Краткие итоги: Введено фундаментальное понятие конструктивного действительного числа. Дано уточнение понятия алгоритма с помощью машины Поста. С помощью объектно-ориентированный подхода реализованы классы конструктивных действительных чисел.

8. Объектно-ориентированнная реализация числовых функций

Пусть X и Y — произвольные множества. Мы под **абстрактной функцией** F будем понимать правило, которое элементу множества X сопоставляет элемент множества Y. Записывается это в виде

$$F : X \to Y.$$

Мы будем рассматривать только однозначные функции. В противном случае всегда можно вместо множества Y рассматривать подмножества множества Y. Функция F может быть определена не на всем множестве X, а только на некотором (непустом) подмножестве

$D \subset X$. Множество D называется **областью определения функции** F.

Мы будем рассматривать функции F, определенные на подмножествах $\mathbb{R}$. Однако в качестве множества Y у нас могут выступать различные множества. Для этого создадим универсальный класс, который сможет представлять любые другие объекты.

```
class TElement
{
    object A; // место для любого объекта
    public TElement(object A)
    {
        SetA(A);
    }
    public object GetA()
    {
        return A;
    }
    public void SetA(object A)
    {
        this.A = A;
    }
}
```

Заметим, что мы воспользовались типом *object*. Этот тип является родительским для всех объектов C# , поэтому возможно в классе $TElement$ хранить ссылку на любой объект (элемент множества Y).

Абстрактный класс представляющий функцию, заданную на некотором отрезке $[a, b]$, выглядит следующим образом

```
    abstract class TAFunc
    {
```

```
protected double a, b;

public TAFunc(double a, double b)
{
    this.a = a;
    this.b = b;
}

public double Get_a()
{
    return a;
}

public double Get_b()
{
    return b;
}

public TElement Calc(double x)
{
    if ((x < a) || (x > b))
    {
        return null;
    }
    else
    {
        return CalcVal(x);
    }
}

protected abstract TElement CalcVal(double x);
```

```
}
```

Для реализации заданной функции, необходимо создать класс наследник, в котором нужно переопределить метод $CalcVal$. Для случая числовых функций, в которых множество Y также является $\mathbb{R}$, мы определим более удобный класс, наследник от $TAFunc$.

```
abstract class TRFunc : TAFunc
{
    public TRFunc(double a, double b) : base(a, b) { }

    public double CalcY(double x)
    {
        return (double)Calc(x).GetA();
    }
}
```

В этом классе метод $CalcY$ возвращает тип *double*, а не *object*, поэтому нет необходимости в преобразовании типов.

Покажем как выглядит класс для описания функции *sin*.

```
class TSinFunc : TRFunc
{
    public TSinFunc(double a, double b) : base(a, b) { }
    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(Math.Sin(x));
        return res;
    }

}
```

Теперь покажем, как с помощью объектно-ориентированнного подхода можно выполнять различные операции над функциями. Созда-

дим класс, который будет представлять собой производную от исходного класса-функции.

```
class TDiffFunc : TRFunc
{
    protected TRFunc Func;
    double h;

    public TDiffFunc(TRFunc Func, double h)
    : base(Func.Get_a(), Func.Get_b())
    {
        this.Func = Func;
        this.h = h;
    }

    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(0);
        double D;

        if ((x + h) <= b)
        {
            D = (Func.CalcY(x + h) - Func.CalcY(x)) / h;
        }
        else
        {
            D = (Func.CalcY(x) - Func.CalcY(x - h)) / h;
        }

        res.SetA(D);
        return res;
```

```
    }
}
```

Теперь с использованием наших классов мы вычислим производные от функции $y = \sin(x)$.

```
TSinFunc Sin = new TSinFunc(0, 2.0 * Math.PI);

TDiffFunc DSin = new TDiffFunc(Sin, 0.00001);
TDiffFunc D2Sin = new TDiffFunc(DSin, 0.0001);

double x0 = Math.PI / 2.0;

Console.WriteLine("DSin({0}) = {1}", x0, DSin.CalcY(x0));
Console.WriteLine("D2Sin({0}) = {1}", x0, D2Sin.CalcY(x0));
```

При выполнении этого кода мы получим:

```
DSin(1.5707963267949) = -5.00000041370186E-06
D2Sin(1.5707963267949) = -0.999999971718068
```

Что близко к точным значениями 0 и -1.

Мы реализовали класс для представления функций, убедились, что наши классы в состоянии выполнять различные действия с функциями. Однако в наших вычислениях мы использовали тип *double* для представления действительных чисел. Этот подход означает, что мы рассматриваем лишь подмножество рациональных чисел. Но из прошлой лекции мы знаем, что на компьютере можно реализовать и большее — машина Поста (а значит теоретически и любой компьютер) может оперировать конструктивными действительными числами. Поэтому естественно рассматривать функции как отображение одного конструктивного действительного числа в другое конструктивное действительное число. В конструктивной математике есть

фундаментальное понятие — **конструктивная функция**. Однако, как известно, единого определения конструктивной функции в строгих рамках конструктивной математики не существует.

Мы дадим определение функции Маркова. Но сначала мы должны ответить на естественный вопрос — а какие конструктивные действительные числа называются равными? Если исходить из того, что конструктивно действительное число — это пара алгоритмов, то из неравенства этих алгоритмов еще не следует неравенство конструктивных действительных чисел. Поэтому мы будем говорить, что x равно y и писать $x = y$, если существует такая вычислимая функция $r : \mathbb{Q} \to \mathbb{N}$, что

$$|x(n) - y(n)| < \varepsilon,$$

для всех $n > r(\varepsilon)$.

Пусть $\mathcal{A}$ — некоторое множество конструктивных действительных чисел такое, что если $x \in \mathcal{A}$ и y — другое конструктивное действительное число и $x = y$, то $y \in \mathcal{A}$. Конструктивной функцией или функцией Маркова называется такой алгоритм $f(x)$, что

1. Для любого $x \in \mathcal{A}$ определено $f(x)$ и $f(x)$ есть конструктивное действительное число.

2. Если x и y суть конструктивные действительные числа, $x \in \mathcal{A}$ и $x = y$, то $f(x) = f(y)$.

Ключевые термины

Абстрактная функция — однозначное отображение одно множества в другое.

Конструктивная функция — алгоритм сопоставляющий одному конструктивному действительному числу другое конструктивное действительное число.

Область определения функции — множество на, котором определена функция.

Краткие итоги: Реализован абстрактный класс на языке C# ,

реализующий функции. Разработаны классы, реализующие конкретные числовые функции и операции над ними.

Глава IV

Объектно-ориентированное моделирование операторных уравнений

9. Объектно-ориентированный подход в моделировании функциональных пространств

В современной математике часто изложение построено на использовании функциональных пространств и методов функционального анализа. Среди фундаментальных понятий функционального анализа мы рассмотрим: метрические пространства, банаховы пространства и гильбертовы пространства. Мы будем предполагать, что понятие линейного пространства является известным.

Метрическим пространством называется пара $(\mathbb{X}, \varrho)$, где $\mathbb{X}$ некоторое множество произвольной природы, а ϱ числовая функция

$$\varrho : \mathbb{X} \times \mathbb{X} \to \mathbb{R},$$

удовлетворяющая следующим условиям для любых $x, y, z \in \mathbb{X}$:

- $\varrho(x, y) \geq 0$

- $\varrho(x, y) = 0$ тогда и только тогда, когда $x = y$
- $\varrho(x, z) \leq \varrho(x, y) + \varrho(y, z)$

В этом случае функция ϱ называется **метрикой** или расстоянием.

Метрика позволяет ввести фундаментальное понятие сходимости в метрическом пространстве. Пусть задано метрическое пространство $(\mathbb{X}, \varrho)$, которое мы будем обозначать $\mathbb{X}_\varrho$ или просто $\mathbb{X}$, если это не приводит к недоразумению. Последовательность x_n сходиться к элементу x_0, если для любого $\varepsilon > 0$ существует такое $N = N(\varepsilon)$, что

$$\varrho(x_n, x_0) < \varepsilon,$$

для всех $n > N$. В этом случае пишут

$$x_0 = \lim_{n\to\infty} x_n.$$

Последовательность x_n называется фундаментальной, если для любого $\varepsilon > 0$ существует такое $N = N(\varepsilon)$, что

$$\varrho(x_p, x_q) < \varepsilon,$$

для всех $p, q > N$. Если метрическое пространство такое, что любая фундаментальная последовательность имеет предел, то такое пространство называется полным метрическим пространством.

Приведем некоторые примеры метрических пространств.

Пространство $\mathbb{R}^n$ с евклидовой метрикой

$$\varrho(x, y) = |x - y| = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}$$

является полным метрическим пространством.

Пространство непрерывных на $[a, b]$, $-\infty < a < b < \infty$ функций $C[a, b]$ является полным метрическим пространством с метрикой

$$\varrho(f, g) = \max_{x\in[a,b]} |f(x) - g(x)|.$$

Это пространство в отличии от $\mathbb{R}^n$ является бесконечномерным функциональным пространством.

Во множестве непрерывных на $[a, b]$ функций можно ввести другую метрику

$$\varrho(f, g) = \int_a^b |f(x) - g(x)|dx.$$

Это метрическое пространство мы обозначим через $L[a, b]$. Пространство $L[a, b]$ не является полным. Например последовательность $f_n(x) = x^n$ является фундаментальной на в $L[0, 1]$, но не имеет предела в пространстве $L[0, 1]$.

Реализуем теперь абстрактное метрическое пространство на C# .

```
abstract class TMetricSpace
{
    public abstract double rho(object x, object y);
}
```

Это класс-заготовка для произвольного метрического пространства. Реализуем метрическое пространство $\mathbb{R}^n$. Для этого сначала нам потребуется создать класс, представляющий n-мерный вектор.

```
class TRn
{
    double[] A;
    int N = 0;

    public TRn(int N)
    {
        this.N = N;
        A = new double[N + 1];
    }
    public double this[int index]
```

```
    {
        get { return A[index]; }
        set { A[index] = value; }
    }
    public int GetN()
    {
        return N;
    }
}
```

А теперь создадим класс являющийся наследником от $TMetricSpace$.

```
class TRnMetric : TMetricSpace
{
    public override double rho(object x, object y)
    {
        TRn X, Y;

        X = (TRn)x;
        Y = (TRn)y;

        double res = 0;

        for (int i = 1; i <= X.GetN(); i++)
        {
            res += (X[i] - Y[i]) * (X[i] - Y[i]);
        }

        return Math.Sqrt(res);
    }
}
```

Проверим «в бою» наши классы. Попробуем вычислить длину диагонали единичного куба.

```
TRn A = new TRn(3);
TRn B = new TRn(3);

A[1] = 0;
A[2] = 0;
A[3] = 0;
B[1] = 1;
B[2] = 1;
B[3] = 1;

TRnMetric M = new TRnMetric();

Console.WriteLine("R = {0}", M.rho(A, B));
```

Запустим и увидим:

```
R = 1.73205080756888
```

Длина диагонали единичного куба, как раз, и равна $R = \sqrt{3}$.

Среди метрических пространств есть такие пространства, в которых можно ввести не только расстояние между элементами, но и длину каждого элемента. Эта длина называется **нормой**. Линейное пространство $\mathbb{X}$ называется **нормированным**, если в нем можно ввести норму для каждого элемента, обозначаемую $\|x\|_{\mathbb{X}}$, удовлетворяющую следующим условиям для всех $x, y \in \mathbb{X}$ и $\lambda \in \mathbb{C}$

- $\|x\| \geq 0$
- $\|x\| = 0$ тогда и только тогда, когда $x = 0$
- $\|\lambda x\| = |\lambda| \|x\|$

- $\|x + y\| \leq \|x\| + \|y\|$

Сразу заметим, что любая норма порождает в нормированном пространстве метрику по формуле

$$\varrho(x, y) = \|x - y\|.$$

Нормированное пространство, являющаяся полным, называется **банаховым пространством**. Рассмотренные ранее примеры метрических пространств являются также и нормированными пространствам:

$$\|x\|_{\mathbb{R}^n} = \sqrt{x_1^2 + \cdots + x_n^2},$$

$$\|f\|_{C[a,b]} = \max_{x \in [a,b]} |f(x)|,$$

$$\|f\|_{L[a,b]} = \left(\int_a^b |f(x)|^2 dx \right)^{1/2}.$$

Соответственно пространства $\mathbb{R}^n$ и $C[a, b]$ являются банаховыми пространствами, а пространство $L[a, b]$ только нормированным.

Посмотрим как на C# выглядят нормированные (банаховы) пространства.

```
abstract class TNormSpace
{
    public abstract double norm(object x);
}

class TCNormSpace : TNormSpace
{
    public override double norm(object x)
    {
        double res = 0;
```

```
        TRFunc F = (TRFunc)x;

        double t = 0;

        // задаем шаг для расчета max
        double dt = (F.Get_b() - F.Get_a()) / 1000;

        while (t < F.Get_b())
        {
            if (Math.Abs(F.CalcY(t)) > res)
            {
                res = Math.Abs(F.CalcY(t));
            }

            t += dt;
        }

        return res;
    }
}
```

Мы реализовали абстрактное нормированное пространство $TNormSpace$, а потом реализуем класс-наследник для пространства $C[a, b]$. Чтобы вычислить нормы некоторых функций реализуем эти функции.

```
    class TXSinFunc : TRFunc
    {
        public TXSinFunc(double a, double b) : base(a, b) { }
        protected override TElement CalcVal(double x)
        {
            TElement res = new TElement(x * Math.Sin(x));
            return res;
```

```
        }

    }

    class TXXFunc : TRFunc
    {
        public TXXFunc(double a, double b) : base(a, b) { }
        protected override TElement CalcVal(double x)
        {
            TElement res = new TElement(x * (1 - x));
            return res;
        }

    }
```

Теперь посчитаем нормы этих функций.

```
    TXSinFunc XSin = new TXSinFunc(0, 2.0 * Math.PI);

    TXXFunc XX = new TXXFunc(0, 1);

    TCNormSpace C = new TCNormSpace();

    Console.WriteLine("||x*sin(x)||={0}", C.norm(XSin));
    Console.WriteLine("||x*(1-x)||={0}", C.norm(XX));
```

После запуска мы увидим

```
||x*sin(x)||=4.81446969901559

||x*(1-x)||=0.25
```

Чтобы вычислять метрику между двумя функциями в пространстве с помощью нормы реализуем еще один класс, который будет представлять разность двух числовых функций.

```
class TSubFunc : TRFunc
{
    TRFunc X, Y;
    public TSubFunc(TRFunc X, TRFunc Y)
    : base(X.Get_a(), X.Get_b())
    {
        this.X = X;
        this.Y = Y;
    }

    protected override TElement CalcVal(double x)
    {
        throw new NotImplementedException();
    }

    public override double CalcY(double x)
    {
        return X.CalcY(x) - Y.CalcY(x);
    }
}
```

Теперь вычислим расстояние между функциями $\sin(x)$ и $\cos(x)$ в пространстве $C[a, b]$

```
TCNormSpace C = new TCNormSpace();

TSinFunc Sin = new TSinFunc(0, 2.0 * Math.PI);
TCosFunc cos = new TCosFunc(0, 2.0 * Math.PI);

TSubFunc SinCos = new TSubFunc(Sin, cos);

Console.WriteLine("||sin(x) - cos(x)||={0}", C.norm(SinCos));
```

В результате получим

```
||sin(x) - cos(x)||=1.41421356237309
```

Таким образом мы видим, что расстояние между этими функциями равно $\sqrt{2}$. И это при том, что норма каждой из функций равна 1!

Однако в некоторых функциональных пространствах можно ввести не только расстояние между функциями, но и угол. Эти пространства называются предгильбертовым. Линейное пространство $\mathbb{X}$ называется **предгильбертовым**, если в этом пространстве можно ввести **скалярное произведение** $(x, y)_{\mathbb{X}}$, которое должно удовлетворять следующим условиям для любых $x, y, z \in \mathbb{X}$ и $\lambda, \mu \in \mathbb{C}$

- $(\lambda x + \mu y, z) = \lambda(x, z) + \mu(y, z)$
- $(x, y) = \overline{(y, x)}$
- $(x, x) \geq 0$, причем $(x, x) = 0$ тогда и только тогда, когда $x = 0$

Предгильбертово пространство является также и нормированным и, соответственно, метрическим. Скалярное произведение порождает норму по формуле

$$\|x\| = (x, x)^{1/2}.$$

Полное предгильбертово пространство называется **гильбертовым**.

Конечномерное пространство $\mathbb{R}^n$ является гильбертовым со скалярным произведением

$$(x, y)_{\mathbb{R}^n} = x_1 y_1 + \cdots + x_n y_n.$$

В пространстве $L[a, b]$ также можно ввести скалярное пространство по формуле

$$(f, g)_{L[a,b]} = \int_a^b f(x)\overline{g(x)}dx.$$

Однако это пространство не будет полным, и, соответственно, гильбертовым.

Для двух ненулевых элементов вещественного предгильбертова пространства можно ввести понятие угла φ. Пусть $x \neq 0$ и $y \neq 0$, тогда углом между этими элементами является величина

$$\varphi = \arccos \frac{(x, y)}{\|x\|\|y\|}.$$

Неравенство Коши-Буняковского, верное в любом предгильбертовом пространстве, гарантирует, что arccos существует. Действительно, согласно этому неравенству

$$|(x, y)| \leq \|x\|\|y\|.$$

Создадим абстрактный класс для предгильбертовых пространств и реализуем класс для пространства $L[a, b]$.

```
abstract class THSpace
{
    public abstract double inner(object x, object y);
}

class TLSpace : THSpace
{
    public override double inner(object x, object y)
    {
        double res = 0;

        TRFunc F = (TRFunc)x;
        TRFunc G = (TRFunc)y;

        double t = 0;
        double dt = (F.Get_b() - F.Get_a()) / 10000;
```

```
            while (t + dt < F.Get_b())
            {
                res += F.CalcY(t) * G.CalcY(t) * dt;

                t += dt;
            }

            return res;
        }
    }
```

Теперь вычислим некоторые скалярные произведения.

```
TLSpace LSpace = new TLSpace();

Console.WriteLine("(sin x, cos x) = {0}", LSpace.inner(Sin, Cos));
Console.WriteLine("(sin x, sin x) = {0}", LSpace.inner(Sin, Sin));
Console.WriteLine("(x(1-x), sin x) = {0}", LSpace.inner(XX, Sin));
```

В результате получим:

```
(sin x, cos x) = 3.94784157532637E-07

(sin x, sin x) = 3.14159265334141

(x(1-x), sin x) = 0.0779244027546068
```

А теперь посчитаем углы между этими функциями. Для этого добавим в класс $THSpace$ метод для подсчета угла.

```
abstract class THSpace
{
    public abstract double inner(object x, object y);
```

```
    public double Angle(object x, object y)
    {
        return Math.Acos(inner(x, y) /
        Math.Sqrt((inner(x, x) * inner(y, y))));
    }
}
```

Теперь выполним следующий код.

```
    Console.WriteLine("Angle(sin x, cos x) = {0}",
    LSpace.Angle(Sin, Cos));
    Console.WriteLine("Angle(sin x, sin x) = {0}",
    LSpace.Angle(Sin, Sin));
    Console.WriteLine("Angle(x(1-x), sin x) = {0}",
    LSpace.Angle(XX, Sin));
```

И получим следующий результат.

```
Angle(sin x, cos x) = 1.57079620111863
Angle(sin x, sin x) = 0
Angle(x(1-x), sin x) = 1.32760476887546
```

Мы видим, что угол между $\sin(x)$ и $\cos(x)$ равен $\frac{\pi}{2} = 90°$, для таких элементов говорят, что они ортогональны. А угол между $\sin(x)$ и $\sin(x)$ равен нулю, что предсказуемо.

Ключевые термины

Банахово пространство — полное нормированное пространство.

Гильбертово пространство — полное предгильбертово пространство.

Метрика — функция на паре элементов метрического пространства, соответствующий аналог расстояния между элементами.

Метрическое пространство — абстрактное пространство в котором введена метрика.

Норма — функция на элементах нормированного пространства, аналог длинны.

Нормированное пространство — линейное пространство, в котором введена норма.

Предгильбертово пространство — линейное пространство, в котором введено скалярное произведение.

Скалярное произведение — функция на паре элементов предгильбертова пространства, удовлетворяющая аксиомам скалярного произведения.

Краткие итоги: Рассмотрены наиболее важные понятия функционального анализа — метрические, нормированные, банаховы, гильбертовы пространства. Даны объектно-ориентированные реализации этих пространств. Проведены вычислительные эксперименты, демонстрирующие функциональные пространства.

10. Объектно-ориентированный подход к реализации линейных операторов

В функциональном анализе большое значения играют линейные операторы. Пусть $\mathbb{X}$ и $\mathbb{Y}$ — линейные пространства, $D \subset \mathbb{X}$ — линейное подпространство. **Линейным оператором** $A : \mathbb{X} \to \mathbb{Y}$ называется такое отображение, что для любых $x, y \in \mathcal{D}$ и $\lambda \in \mathbb{C}$ верно

$$A(\lambda x + y) = \lambda Ax + Ay.$$

Множество $\mathcal{D}$ называется областью определения оператора A, и мы часто будем обозначать $\mathcal{D}(A)$.

Если пространства $\mathbb{X}$ и $\mathbb{Y}$ банаховы пространства, то можно рассматривать **непрерывные операторы**. Оператор A называется непрерывным, если $\mathcal{D}(A) = \mathbb{X}$ и для любой сходящейся в $\mathbb{X}$ последовательности x_n имеет место

$$\lim_{n\to\infty} Ax_n = A \lim_{n\to\infty} x_n.$$

Для любого ограниченного оператора можно ввести конечную норму этого оператора по формуле

$$\|A\| = \sup_{x\in\mathbb{X},x\neq 0} \frac{\|Ax\|_{\mathbb{Y}}}{\|x\|_{\mathbb{X}}} = \sup_{x\in\mathbb{X},\|x\|_{\mathbb{X}}\leq 1} \|Ax\|_{\mathbb{Y}}.$$

Соответственно, любой линейный непрерывный оператор является ограниченным (верно и обратное).

В конечномерном случае, когда $\mathbb{X} = \mathbb{R}^n$, $\mathbb{Y} = \mathbb{R}^n$ линейные операторы являются матрицами $n \times n$. Примером неограниченного оператора является оператор дифференцирования. Пусть $\mathbb{X} = \mathbb{Y} = C[a, b]$, множество D состоит из непрерывных на $[a, b]$ функций, имеющих непрерывную производную, тогда оператор

$$A[f] = \frac{df}{dx}$$

является линейным оператором.

Мы будем рассматривать линейные оператора в бесконечномерных гильбертовых пространствах. Пусть $\mathbb{H}$ — такое гильбертово пространство, что существует бесконечная последовательность линейно независимых элементов $e_k \in \mathbb{H}$ таких, что

$$(e_k, e_l)_{\mathbb{H}} = 0, \quad k \neq l$$

и

$$(e_k, e_k)_{\mathbb{H}} = 1.$$

Также будем предполагать, что любой элемент пространства $\mathbb{H}$ однозначно представляется рядом

$$u = \sum_{k=1}^{\infty} u_k e_k,$$

где ряд сходится по метрике пространства $\mathbb{H}$. Элементы $\{e_k\}$ называются ортонормированным базисом, а сам ряд — **абстрактными**

рядом Фурье. Числа u_k называются коэффициентами Фурье, и для них имеет место следующая формула

$$u_k = (u, e_k)_{\mathbb{H}}.$$

Имеет место следующее равенство, называемое равенство Парсевалем

$$\|u\|_{\mathbb{H}} = \left(\sum_{k=1}^{\infty} |u_k|^2 \right)^{1/2}.$$

Таким образом, чтобы задать любой элемент пространства $\mathbb{H}$ достаточно задать последовательность коэффициентов Фурье, т.е. таких чисел, что

$$\sum_{k=1}^{\infty} |u_k|^2 < \infty.$$

Иногда удобно рассматривать ряды, где $k = 0, 1, \cdots$ или $k = 0, \pm 1, \pm 2, \cdots$. Для этих случаев все аналогично.

Любой оператор в гильбертовом пространстве можно задать как преобразование последовательности коэффициентов Фурье. Для линейного оператора это преобразование должно быть линейным.

Большой интерес представляю такие линейные операторы, которые можно представить последовательностью чисел (вообще говоря, комплексных) α_k, а действие этого оператора на элемент следующим образом. Для любого

$$u = \sum_{k=1}^{\infty} u_k e_k$$

имеем

$$A[u] = \sum_{k=1}^{\infty} \alpha_k u_k e_k.$$

Если имеет место оценка $|\alpha_k| \leq M$, то такой оператор будет ограниченным (и непрерывным). Такие операторы мы будем называть **диагональными**.

Классическим примером гильбертова пространства является пространство $L_2(0, 2\pi)$ — измеримых на $[0, 2\pi]$ функций и имеющих интегрируемый по Лебегу квадрат модуля на $[0, 2\pi]$. Скалярное произведение задается формулой

$$(f, g)_{L_2(0,2\pi)} = \int_0^{2\pi} f(x)\overline{g(x)}dx.$$

Базис в этом пространстве можно задать с помощью следующих функций

$$e_k = \frac{1}{\sqrt{2\pi}} e^{ikx}, \quad k = 0, \pm 1, \pm 2, \cdots .$$

Зададим в это пространстве линейный оператор с помощью последовательности $\alpha_k = ik$, тогда этот оператор будет оператором дифференцирования. Как мы видим, этот оператор не будет ограниченным и будет определен не на всех функциях, а только на тех функциях, на которых будет сходится ряд

$$\sum_{k=0}^{\infty} |\alpha_k u_k|^2 < \infty.$$

С помощью разложения по базису достаточно легко можно вычислять действие оператора. Разумеется, что такое разложение оператора зависит от выбора базиса, который можно выбирать многими способами.

Обратимся к вопросу об объектно-ориентированнной реализации элементов гильбертовых пространств и линейных диагональных операторов.

```
abstract class THElement
{
    abstract public double a(int k);
}
```

```
abstract class THOperator
{
    THElement u;
    public THOperator(THElement u)
    {
        this.u = u;
    }

    abstract protected double alpha(int k);

    public double a(int k)
    {
        return alpha(k) * u.a(k);
    }
}
```

Здесь мы описали два абстрактных класса. Первый $THElement$ предназначен для описания коэффициентов Фурье элементов гильбертовых пространств, а второй класс $THOperator$ описывает диагональный оператор.

Рассмотрим пространство $L_2(0, \pi)$ в котором выберем следующий базис

$$e_k = \sqrt{\frac{2}{\pi}} \sin kx, \quad k = 1, 2, \cdots .$$

Рассмотрим функцию

$$f(x) = x(\pi - x).$$

В разложении по нашему базису эта функция имеет следующий вид

$$f(x) = \sum_{k=1}^{\infty} -\frac{4}{\pi} \frac{(-1)^k - 1}{k^3} \sin kx.$$

Здесь коэффициенты Фурье имеют вид

$$f_k = -2\sqrt{\frac{2}{\pi}} \frac{(-1)^k - 1}{k^3}.$$

Введем в нашем пространстве диагональный оператор D^2 со следующими числами α_k

$$\alpha_k = -k^2.$$

Для вычисления $D^2[f]$ нам даже не потребуется компьютер, можно сразу записать

$$g(x) = \mathcal{D}^2[f](x) = \sum_{k=1}^{\infty} \frac{4}{\pi} \frac{(-1)^{k+1} - 1}{k} \sin kx.$$

Поскольку мы имеем сходящийся ряд

$$\frac{4}{\pi} \sum_{k=1}^{\infty} \frac{1}{k^2} < \infty,$$

то можно утверждать, что функция $f(x) = x(\pi - x)$ принадлежит области определения оператора $\mathcal{D}^2$. А теперь заметим, что разложение функции $h = -2$ по нашему базису имеет вид

$$-2 = \sum_{k=1}^{\infty} \frac{4}{\pi} \frac{(-1)^{k+1} - 1}{k} \sin kx.$$

Поэтому мы получаем, что

$$\mathcal{D}^2 f(x) = -2,$$

Функция -2 является второй производной по переменной x от функции $f(x)$. И это не совпадение — наш оператор $\mathcal{D}^2$, действительно, является оператором дважды дифференцирования. В этом несложно убедиться, если заметить, что

$$\mathcal{D}^2 \sin kx = -k^2 \sin kx.$$

Однако не каждая дважды дифференцируемая функция принадлежит области определения нашего оператора. Например функция $f(x) = 1$ имеет разложение по базису в нашем пространстве

$$1 = \sum_{k=1}^{\infty} -\frac{2}{\pi} \frac{1}{k} ((-1)^k - 1).$$

Коэффициенты Фурье функции $D^2[1]$ имеют вид

$$g_k = \sqrt{\frac{2}{\pi}} k((-1)^k - 1).$$

Однако ряд

$$\sum_{k=1}^{\infty} \frac{2}{\pi} k^2((-1)^k - 1)^2$$

расходится. Дело в том, что область определения оператора $\mathcal{D}^2$ накладывает условия не только на гладкость, но и на краевые условия — необходимо, чтобы функция принимала нулевые значения при $x = 0$ и $x = \pi$.

Проверим наши теоретические выкладки с помощью компьютерного моделирования на C# . Для этого мы реализуем следующие классы.

```
abstract class TL2Element : THElement
{
    public double Calc(double x)
    {
        double res = 0;

        for (int k = 1; k <= 1000; k++)
        {
            res += a(k) * Math.Sin((double)k * x);
        }

        res *= Math.Sqrt(2.0 / Math.PI);

        return res;
    }
}
```

```
class TL2F1 : TL2Element
{
    public override double a(int k)
    {
        double kx = (double)k;

        if ((k % 2) == 0)
        {
            return 0;
        }
        else
        {
            return -2.0 * Math.Sqrt(2.0 / Math.PI) *
            (-2.0) * (1.0 / (kx * kx * kx));
        }
    }
}

class TL2F2 : TL2Element
{
    public override double a(int k)
    {
        double kx = (double)k;

        if ((k % 2) == 0)
        {
            return 0;
        }
        else
        {
            return - Math.Sqrt(2.0 / Math.PI) *
```

```
            (-2.0) * (1.0 / kx);
        }
    }
}

abstract class TL2Operator : THOperator
{
    public TL2Operator(THElement u) : base(u) { }

    public double Calc(double x)
    {
        double res = 0;

        for (int k = 1; k <= 1000; k++)
        {
            res += a(k) * Math.Sin((double)k * x);
        }

        res *= Math.Sqrt(2.0 / Math.PI);

        return res;
    }

}

class TD2Operator : TL2Operator
{
    public TD2Operator(THElement u) : base(u) { }

    public override double alpha(int k)
    {
```

```
            double kx = (double)k;

            return -(kx * kx);
        }
    }
```

Теперь попробуем вычислить значения действия оператора в заданной точке.

```
TL2F1 F1 = new TL2F1();
TL2F2 F2 = new TL2F2();

TD2Operator D2 = new TD2Operator(F1);
Console.WriteLine("D2[x(pi-x)](1) = {0}", D2.Calc(1.0));

D2 = new TD2Operator(F2);
Console.WriteLine("D2[1](1) = {0}", D2.Calc(1.0));
```

В итоге получим:

```
D2[x(pi-x)](1) = -1.99914825586894
D2[1](1) = 425.069483543905
```

Как видим, первое значение довольно точно совпадает с точным, а второе значение показывает, что применять оператор $\mathcal{D}^2$ к функции $f(x) = 1$ нельзя.

Ключевые термины

Абстрактный ряд Фурье — аналог ряда Фурье в гильбертовом пространстве.

Диагональный оператор — оператор, который можно задать путем умножения коэффициентов абстрактного ряда Фурье на числа.

Линейный оператор — линейное отображение одного линейного пространства в другое.

Непрерывный оператор — непрерывное отображение одного метрического пространства в другое.

Краткие итоги: Приведена конструкционная реализация линейных неограниченных операторов в гильбертовом пространстве. Реализованы классы операторов, для представления линейных операторов на языке C# .

11. О решении операторных уравнений

Как мы уже отмечали, многие математические задачи могут быть записаны в виде операторных уравнений. И одной из основных задач при рассмотрении операторных уравнений является нахождение обратного оператора. Мы рассмотрим методы решения операторных уравнений, основанные на **методе Галеркина**. Основным модельным примером в нашем рассмотрении будет **краевая (двухточечная) задача** для обыкновенного дифференциального уравнения второго порядка.

Мы будем рассматривать следующее уравнение

$$-y''(x) + p(x)y'(x) + q(x)y(x) = f(x),\ x \in (0, \pi), \tag{IV.1}$$

со следующими краевыми условиями

$$y(0) = y(\pi) = 0 \tag{IV.2}$$

Коэффициенты $p(x)$ и $q(x)$, а также функция $f(x)$ считаются известными. Однако в этой лекции мы будем рассматривать лишь случай, когда $p(x) \equiv 0$, а $q(x) \equiv q \geq 0$.

Используя результаты предыдущей лекции, мы будем рассматривать следующее **гильбертово пространство** $L_2(0, \pi)$, в котором в качестве базисных функций выбраны функции

$$e_k = \sqrt{\frac{2}{\pi}} \sin kx,\ k = 1, 2, \ldots.$$

Введем еще одно пространство, которое обозначим $H^2(0,\pi)$. Мы будем говорить, что функция $u \in L_2(0,\pi)$ принадлежит пространству $H^2(0,\pi)$, если коэффициенты Фурье этой функции:

$$u = \sum_{k=1}^{\infty} u_k \sqrt{\frac{2}{\pi}} \sin kx$$

удовлетворяют следующему условию — для них сходится ряд:

$$\|u\|^2_{H^2(0,\pi)} = \sum_{k=1}^{\infty} k^2 |u_k|^2 < \infty.$$

Последнее условие требует более быстрого убывания коэффициентов Фурье, что соответствует большей гладкости решения. Как мы видели в прошлой лекции, если функция $u \in H^2(0,\pi)$, то вторая производная этой функции (точнее, действие оператора $\mathcal{D}^2$) принадлежит $L_2(0,\pi)$. Введем еще один простой оператор

$$Q : L_2(0,\pi) \to L_2(0,\pi)$$

умножения на константу q. Очевидно, что это тоже диагональный оператор, для которого коэффициенты α_k равны

$$\alpha_k = q.$$

Тогда задачу (IV.1)–(IV.2) можно записать в операторном виде:

$$-D^2[y] + Q[y] = f, \tag{IV.3}$$

где $f \in L_2(0,\pi)$. Соответственно, решением задачи (IV.3) называется функция $y \in H^2(0,\pi)$, удовлетворяющая уравнению (IV.3).

Операторный подход позволяет довольно просто получить решение задачи (IV.3). Чтобы вывести аналитико-численные формулы, для решения задачи (IV.3) представим функцию f разложением в ряд:

$$f(x) = \sum_{k=1}^{\infty} f_k \sqrt{\frac{2}{\pi}} \sin kx,$$

где коэффициенты Фурье могут быть найдены по формуле

$$f_k = \int_0^\pi f(x)\sqrt{\frac{2}{\pi}}\sin kx dx.$$

Соответственно, решение y мы будем искать в виде:

$$y(x) = \sum_{k=1}^{\infty} y_k \sqrt{\frac{2}{\pi}}\sin kx.$$

Таким образом задача сводится к нахождению чисел y_k. Находить эти числа мы будем следующим образом — подставим разложение по базисным функциям в уравнение (IV.3):

$$-\mathcal{D}^2[\sum_{k=1}^{\infty} y_k \sqrt{\frac{2}{\pi}}\sin kx] + Q[\sum_{k=1}^{\infty} y_k \sqrt{\frac{2}{\pi}}\sin kx] = \sum_{k=1}^{\infty} f_k \sqrt{\frac{2}{\pi}}\sin kx,$$

Что можно записать следующим образом

$$\sum_{k=1}^{\infty} \left(k^2 + q\right) y_k \sqrt{\frac{2}{\pi}}\sin kx = \sum_{k=1}^{\infty} f_k \sqrt{\frac{2}{\pi}}\sin kx$$

Умножая скалярно это равенство на базисные функции $\sqrt{\frac{2}{\pi}}\sin kx$, получаем цепочку (бесконечную) равенств:

$$(k^2 + q)y_k = f_k.$$

От куда получаем (в силу неотрицательности q):

$$y_k = \frac{f_k}{k^2 + q}.$$

Наш операторный подход позволяет получать решение краевой задачи для дифференциального уравнения используя лишь арифметические операции.

Единственный нетривиальный момент при применении нашего операторного подхода состоит в том, чтобы получить разложение

правой части по выбранному базису. Однако поскольку для получения этого разложения нужно вычислять скалярные произведения, то мы можем воспользоваться ранее созданными классами для гильбертовых пространств.

Перейдем к программированию классов для реализации операторного подхода к решению задачи (IV.1)–(IV.2).

```
// класс для вычисления оператора
class TOE {
    TD2Q D2Q;

    public TOE(double q, TRFunc F)
    {
        D2Q = new TD2Q(q, new TL2Right(F));
    }

    public double Y(double x)
    {
        return D2Q.Calc(x);
    }
}

// класс оператора
class TD2Q : TL2Operator {
    double q;
    public TD2Q(double q, THElement u) : base(u)
    {
        if (q < 0)
        {
            this.q = 0;
        }
```

```
        else
        {
            this.q = q;
        }
    }

    public override double alpha(int k)
    {
        double kx = (double)k;

        return 1.0 / (kx * kx + q);
    }
}

// класс - базисные функции
class TE_k : TRFunc
{
    int k;
    public TE_k(int k) : base(0, Math.PI)
    {
        this.k = k;
    }

    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(Math.Sqrt(2.0 / Math.PI) *
        Math.Sin((double)k * x));
        return res;
    }
}
```

```
// класс - правые части уравнения
class TL2Right : TL2Element
{
    protected TLSpace Ld;
    protected TRFunc F;
    public TL2Right(TRFunc F)
    {
        Ld = new TLSpace();
        this.F = F;
    }

    // разложение по базису
    public override double a(int k)
    {
        TE_k e_k = new TE_k(k);
        return Ld.inner(F, e_k);
    }
}

class TRight1 : TRFunc
{
    public TRight1() : base(0, Math.PI) { }

    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(1.0);
        return res;
    }
}

class TRight2 : TRFunc
```

```
{
    public TRight2() : base(0, Math.PI) { }

    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(Math.Sin(x));
        return res;
    }
}

class TRight3 : TRFunc
{
    public TRight3() : base(0, Math.PI) { }

    protected override TElement CalcVal(double x)
    {
        TElement res = new TElement(Math.Cos(x));
        return res;
    }
}
```

Теперь будем решать численно нашу задачу для разных значений q и правых частей $f(x)$.

```
TRight1 F1 = new TRight1();
TOE OE = new TOE(0, F1);

Console.WriteLine("q = 0; f(x)=1; Y(1) = {0}", OE.Y(1.0));

TRight2 F2 = new TRight2();
OE = new TOE(0, F2);
Console.WriteLine("q = 0; f(x)=sin(x); Y(1) = {0}", OE.Y(1.0));
```

```
TRight3 F3 = new TRight3();
OE = new TOE(0, F3);
Console.WriteLine("q = 0; f(x)=cos(x); Y(1) = {0}", OE.Y(1.0));

OE = new TOE(10.0, F1);
Console.WriteLine("q = 10; f(x)=sin(x); Y(1) = {0}", OE.Y(1.0));

OE = new TOE(10.0, F2);
Console.WriteLine("q = 10; f(x)=sin(x); Y(1) = {0}", OE.Y(1.0));

OE = new TOE(10.0, F3);
Console.WriteLine("q = 10; f(x)=cos(x); Y(1) = {0}", OE.Y(1.0));
```

В итоге получим следующие результаты:

```
q = 0; f(x)=1; Y(1) = 1.07079628676603

q = 0; f(x)=sin(x); Y(1) = 0.841470984797919

q = 0; f(x)=cos(x); Y(1) = 0.176922106919835

q = 10; f(x)=sin(x); Y(1) = 0.0956527832996927

q = 10; f(x)=sin(x); Y(1) = 0.0764973622552322

q = 10; f(x)=cos(x); Y(1) = 0.0453741941981012
```

Сравним наши результаты с точными решениями при $q = 0$. Задача

$$-y''(x) = 1,\ x \in (0, \pi),$$

$$y(0) = y(\pi) = 0$$

Имеет точное решение

$$y(x) = -\frac{1}{2}x^2 + \frac{\pi}{2}x,$$

значение $y(1)$ равно

$$y(1) = -\frac{1}{2} + \frac{\pi}{2} = 1.07079632679489661923132169163 98\ldots.$$

Задача

$$-y''(x) = \sin(x),\ x \in (0, \pi),$$

$$y(0) = y(\pi) = 0$$

Имеет точное решение

$$y(x) = \sin(x)$$

значение $y(1)$ равно

$$y(1) = 0.84147098480789650665250232163 03\ldots.$$

Задача

$$-y''(x) = \cos(x),\ x \in (0, \pi),$$

$$y(0) = y(\pi) = 0$$

Имеет точное решение

$$y(x) = \cos x + \frac{2}{\pi}x - 1$$

значение $y(1)$ равно

$$y(1) = 0.17692207823572106047647166093303\ldots.$$

Мы видим, что наши численные результаты довольно точны. Однако если бы мы использовали аналитические выражения для коэффициентов Фурье, то точность и скорость вычислений были бы значительно выше. Это выражает тот факт, что использование теоретических

знаний при конструировании численных методов могут значительно увеличить эффективность последних.

Ключевые термины

Гильбертово пространство $L_2(0, \pi)$ — наиболее известное гильбертово функциональное пространство, состоящее из измеримых функций, интегрируемых в квадратом модуля.

Краевая задача — дифференциальное уравнение с заданными условиями на решение на концах отрезка.

Метод Галеркина — общий метод для приближенного нахождения решений операторных уравнений.

Краткие итоги: Подробно рассмотрены методы численного решения краевой задачи для дифференциального уравнения второго порядка. С помощью реализованных классов, проведены различные вычислительные эксперименты.

Часть II

Базовые численные методы

Глава V

Решение уравнений

12. Линейные уравнения

Задачи линейно алгебры относятся основным методам вычислительной математики. Это обусловлено тем, что линейные модели играют первостепенную роль, а их численная реализация требует решать задачи линейной алгебры. В отличии от задач математической физики задачи линейной алгебры являются конечномерными. К основным задачам линейной алгебры можно отнести задачи:

1. Решения систем линейных алгебраических уравнений.
2. Нахождение обратных матриц, а также приведение матриц к каноническому виду (диагональному или к форме Жордана).
3. Нахождение собственных значений и собственных функций матриц.

Мы рассмотрим первую наиболее часто встречающуюся задачу нахождения решений систему линейных алгебраических уравнений с невырожденной квадратной матрицей. Матрица называется **невырожденной**, если ее определитель не равен нулю.

Будем рассматривать квадратную матрицу

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Пусть также нам задан вектор-столбец

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

В матричной форме система линейных алгебраических уравнений может быть записана следующим образом

$$Ax = b, \tag{V.1}$$

где неизвестным является вектор-столбец

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

В развернутой форме задача (V.1) может быть записана следующим образом

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ &\dots \\ &a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{aligned}$$

Из результатов линейной алгебры следует, что если матрица A неособая, т.е. $\det A \neq 0$, то задача (V.1) имеет единственное решение для любой правой части b. Однако на практике нахождение решений даже матриц небольшой размерности может составлять большую проблему. Дело в том, что если рассмотреть систему 2×2, то решение

этой системы имеет геометрический смысл точки пересечения двух прямых, заданный первым и вторым уравнением. Легко себе представить ситуацию, когда угол пересечения этих прямых может быть очень малым.

Для количественной оценки «сложности» решения задачи существует понятие **числа обусловленности матрицы** A. Для неособой матрицы A число обусловленности вводится по формуле

$$\mu(A) = \|A\|\|A^{-1}\|,$$

где $\|A\|$ — норма матрицы, которая вводиться для матрицы A, как для ограниченного линейного оператора в пространстве $\mathbb{R}^n$. Можно показать, что число обусловленности μ всегда превосходит единицу. Чем больше это число, тем «сложнее» будет решить задачу нахождение решения этой задачи.

При решении задач нахождения решения системы линейных алгебраических уравнений можно выделить два подхода: точные методы нахождения решений и итерационные методы. Сначала рассмотрим «точные» методы решения системы линейных алгебраических уравнений. Слово «точные» мы пишем в кавычках, поскольку хотя нахождение решения можно осуществить этими методам за конечное число операций, однако среди этих операций обязательно есть операции деления а, как мы уже отмечали, операция деления не всегда может быть реализованы точно.

Главным методом для прямого нахождения решений системы линейных алгебраических уравнений является **метод Гаусса**. Этот метод иногда называют методом исключения неизвестных. Суть этого метода состоит в том, чтобы сначала из первого уравнения выразить x_1 через коэффициенты матрицы, правую часть и x_i, $i = 2, 3, \ldots, n$. Подставив это выражение для x_1 в остальных уравнениях, можно получить другую систему линейных алгебраических уравнений порядка $(n-1) \times (n-1)$. Таким образом можно получить одно линейное

уравнение относительно x_n, которое решается тривиально. Вычислив x_n через коэффициенты матрицы A и правую часть b, используя полученные ранее выражения для x_{n-1} через x_n, x_{n-2} через x_{n-1} и x_n и т.д., можно вычислить все x_i, $i = 1, \ldots, n$. Условием для выполнения этого метода является возможность осуществлять выражение x_i. Если матрица A является неособой, то условие осуществимости метода Гаусса всегда выполнено.

Следует отметить, что решение системы линейных алгебраических уравнений с полной матрицей, т.е. когда матрица не имеет специальной структуры заполнения, представляет собой довольно сложную проблему. К счастью, часто в задачах возникают системы уравнений с матрицами специального вида. Для этих случаев существуют отдельные методы, которые могут быть значительно эффективнее метода Гаусса.

Часто возникают уравнения с симметричной положительно определенной матрицей. Для таких уравнений мы рассмотрим **метод Холецкого**. Пусть наша система уравнений задана с помощью симметричной положительно определенной матрицей A. Метод Холецкого состоит в том, чтобы представить матрицу A в виде

$$A = LL^T,$$

где L — нижняя треугольная матрица

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix}$$

Если мы построим разложение $A = LL^T$, то уравнение (V.1) может быть заменено двумя уравнениями, которые можно последовательно решить

$$Ly = b, \quad L^T x = y.$$

Обращение треугольной матрицы проблемы не составляет и может быть осуществлено по рекуррентным формулам. Метод Холецкого состоит в том, что элементы матрицы L находятся последовательно по формулам

$$l_{11} = (a_{11})^{1/2},$$

$$l_{i1} = a_{i1}/l_{11}, \qquad i = 2, 3, \ldots, n;$$

$$\ldots$$

$$l_{kk} = (a_{kk} - l_{k1}^2 - l_{k2}^2 - \cdots - l_{k,k-1}^2)^{1/2},$$

$$l_{ik} = (a_{ik} - l_{i1}l_{k1} - l_{i2}l_{k2} - \cdots - l_{i,k-1}l_{k,k-1})/l_{kk}, \quad i = k+1, \ldots, n;$$

$$\ldots$$

$$l_{mm} = (a_{mm} - l_{m1}^2 - l_{m2}^2 - \cdots - l_{m,m-1}^2)^{1/2}.$$

В приложениях часто возникают другие уравнения специального вида. В этих уравнениях матрица является трехдиагональной. Для таких уравнений существует очень эффективный алгоритм, называемый **методом прогонки**.

Пусть нам необходимо решить следующее уравнение

$$a_0 z_0 + c_0 z_1 = d_0$$

$$b_n z_{n-1} + a_n z_n + c_n z_{n+1} = d_n, \ n = 1, \ldots, N-1 \qquad \text{(V.2)}$$

$$b_N z_{N-1} + a_N z_N = d_N$$

Сначала мы определяем прогоночные коэффициенты α_n и β_k согласно следующим рекуррентным соотношениям

$$\alpha_0 = -\frac{c_0}{a_0}, \quad \beta_0 = \frac{d_0}{a_0}$$

$$\alpha_n = -\frac{c_n}{a_n + b_n\alpha_{n-1}}, \quad \beta_n = \frac{d_n - b_n\beta_{n-1}}{a_n + b_n\alpha_{n-1}}, \quad n = 1, 2, \ldots, N-1.$$

После вычисления прогоночных коэффициентов можно рекуррентно вычислить и решения уравнения (V.2)

$$z_N = \frac{d_N - b_N \beta_{N-1}}{a_N + b_N \alpha_{N-1}},$$

$$z_n = \alpha_n z_{n+1} + \beta_n, \quad n = N-1, N-2, \ldots, 0.$$

Приведем класс на языке C# , который будем обращать трехдиагональную матрицу методом прогонки.

```
class TProgon
{
    public double[] a;
    public double[] b;
    public double[] c;
    public double[] d;

    public double[] z;

    double[] alpha;
    double[] beta;

    public void CalcZ()
    {
        int N = a.Count();

        alpha = new double[N];
        beta = new double[N];

        z = new double[N + 1];

        alpha[0] = -c[0] / a[0];
        beta[0] = d[0] / a[0];
```

```
        int n;

        for (n = 1; n < N; n++)
        {
            alpha[n] = -(c[n]) / (a[n] + b[n] * alpha[n - 1]);

            beta[n] = (d[n] - b[n] * beta[n - 1]) /
             (a[n] + b[n] * alpha[n - 1]);
        }

        z[N] = (d[N] - b[N] * beta[N - 1]) /
        (a[N] + b[N] * alpha[N - 1]);

        for (n = N - 1; n <= 0; n--)
        {
            z[n] = alpha[n] * z[n + 1] + beta[n];
        }
    }
}
```

Апробацию нашего класса мы проведем в лекции, посвященной кубическим сплайнам. При построении сплайнов возникает необходимость решать систему алгебраических уравнений с трехдиагональной матрицей.

Ключевые термины

Невырожденная матрица — квадратная матрица с определителем отличным от нуля.

Число обусловленности матрицы — числовая характеристика вычислительной сложности обращения матрицы.

Метод Гаусса — основной метод нахождения решений системы линейных алгебраических уравнений с помощью последовательного исключения неизвестных.

Метод Холецкого — метод для обращения симметричных положительно определенных матриц.

Метод прогонки — метод для обращения трехдиагональной матрицы.

Краткие итоги: Рассмотрены численные методы решения задач линейной алгебры. Приведены различные методы, соответствующие матрицам специального вида.

13. Нелинейные уравнения

Решение нелинейных уравнений является одной из самых распространенных задач математики. В лекции, посвященной особенностям вычислительных процедур, мы рассматривали простейший метод решения скалярного уравнения для вычисления корня квадратного из 2. Этот метод — половинного деления имеет много достоинств. Главным достоинством этого метода является то, что для приближенных решений автоматически получается оценка точности. Второе достоинство данного метода состоит в том, что этот метод легко реализуем. Однако у этого метода есть и недостатки. Для реализации этого метода мы должны знать отрезок, на концах которого функция принимает значения разного знака. А также этот метод является сугубо скалярным и не может быть обобщен на многомерный случай.

Сначала рассмотрим задачу о нахождении корня одного уравнения

$$f(x) = 0.$$

Если предположить, что функция $f(x)$ является дифференцируемой, и ее производная есть непрерывная функция $f'(x)$, то находить приближенные решения возможно с помощью **метода Ньютона**.

Формулы метода Ньютона получаются из следующих соображений. Пусть x^* — **корень функции** f, а x_n некоторое приближение к этому корню. Тогда имеем

$$f(x^*) = f(x_n + (x^* - x_n)) = f(x_n) + (x^* - x_n)f'(\xi) = 0.$$

Если вместо неизвестной точки ξ взять значение x_n, то следующее приближение можно вычислять по формулам

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Можно показать, что если **начальное приближение** x_0 достаточно близко к искомому корню, то итерации, вычисленные по методу Ньютона сходятся к этому корню. К сожалению, это условие носит скорее теоретический характер, поскольку не понятно, что значит «достоаточно близко» и как это условие проверить. Другим достаточным условием сходимости метода Ньютона является условие

$$|f(x)f'(x)| < (f'(x))^2.$$

Также из геометрических соображений можно получить, что итерации сходятся к корню с той стороны, с которой выполнено условие

$$f(x)f''(x) \geq 0,$$

если функция f дважды дифференцируема.

Метод Ньютона сходится достаточно быстро, однако у него есть недостатки, состоящие в том, что факт сходимости метода и его скорость сходимости зависят от начального приближения. Вторым серьезным недостатком этого метода является то, что для его реализации нужно знать аналитическое значение производной.

Для устранения последнего недостатка был разработан **метод секущих**, который иногда называется методом хорд. В этом методе производная заменяется конечной разностью

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

Тогда итерации по методу секущих вычисляются по формулам

$$x_{n+1} = x_n - \frac{(x_n - x_{n-1})f(x_n)}{f(x_n) - f(x_{n-1})}.$$

Для реализации метода секущих необходимо знать два предыдущих значения x_n и x_{n-1}. Такие методы называются двухшаговыми. Метод секущих сходиться медленнее, чем метод Ньютона, зато этот метод не требуем знания производной функции $f(x)$.

Реализуем класс для вычислений по методу секущих.

```
class TSolve
{
    int Max_Iter;
    double Eps;
    TRFunc f;

    public TSolve(int Max_Iter, double Eps, TRFunc f)
    {
        this.Max_Iter = Max_Iter;
        this.Eps = Eps;
        this.f = f;
    }

    public double Solve(double x0, double x1)
    {
        double xn1 = 0;
        double xn = x1;
        double xn_1 = x0;

        for (int n = 2; n <= Max_Iter; n++)
        {
            if (Math.Abs(f.CalcY(xn) - f.CalcY(xn_1)) < Eps)
```

```
            {
                break;
            }

            xn1 = xn - ((xn - xn_1) * f.CalcY(xn)) /
            (f.CalcY(xn) - f.CalcY(xn_1));

            Console.WriteLine("x_{0} = {1}", n, xn1);

            xn_1 = xn;
            xn = xn1;
        }

        return xn1;
    }
}
```

В этом классе мы используем класс $TRFunc$, реализующий числовые функции. Для начала вычислений необходимо задать два значения x_0 и x_1. При инициализации класса следует определить два параметра — максимальное количество итераций Max_Iter и минимальную невязку Eps. Вычисления останавливаются в двух условиях — либо когда превышено количество итераций либо, когда выполнится условие

$$|f(x_n) - f(x_{n-1})| < Eps.$$

Проведем тестирование нашего метода.

```
TSinFunc Sin = new TSinFunc(-100, 100);
TSolve Solve = new TSolve(10, 1E-12, Sin);

Console.WriteLine("x_0 = {0}\tx_1 = {1}", 1, 1.5);
Solve.Solve(1, 1.5);
```

```
    Console.WriteLine("\n\nx_0 = {0}\tx_1 = {1}", 1.7, 1.8);
    Solve.Solve(2, 2.2);
```

Мы будем вычислять корни функции $f(x) = \sin x$. В результате двух запусков нашего класса мы получим следующие результаты

```
x_0 = 1 x_1 = 1.5

x_2 = -1.69660749346521

x_3 = -0.102640761680463

x_4 = 0.0809387254524748

x_5 = -3.00669920823837E-05

x_6 = 3.28414216333597E-08

x_7 = -4.94283429392287E-18

x_8 = 7.70371977754894E-34


x_0 = 1.7  x_1 = 1.8

x_2 = 3.80414325114683

x_3 = 3.111014449759

x_4 = 3.14383344379986
```

```
x_5 = 3.14159232994785

x_6 = 3.14159265359006

x_7 = 3.14159265358979
```

Видно, что если рассматриваемая функция имеет несколько корней, то различные значения начальных приближений могут приводить к различным решениям нашего уравнения.

Рассмотрим теперь вопрос о решении системы трансцендентных уравнений. Пусть необходимо решить следующую систему уравнений

$$f_k(x_1, x_2, \ldots, x_n) = 0, \quad 1 \leq k \leq n. \tag{V.3}$$

Как правило такие системы решают методом простых итераций. Для этого необходимо систему (V.3) записать в виде

$$x_k = \varphi_k(x_1, x_2, \ldots, x_n), \quad 1 \leq k \leq n. \tag{V.4}$$

Или в векторной форме

$$x = \varphi(x),$$

где $\varphi : \mathbb{R}^n \to \mathbb{R}^n$. Вместо нахождение корня системы (V.3) мы пришли к задаче нахождения неподвижной точки.

Метод простых итераций выглядит довольно просто. Необходимо задать начальное приближение x^0, а дальнейшие вычисления проводить по схеме

$$x^{n+1} = \varphi(x^n).$$

Однако успех этого метода зависит от свойств функции φ и начального приближения. Приведем одно достаточное условие сходимости метода простых итераций для случая, когда функции φ_k являются непрерывно дифференцируемыми. Построим матрицу M с элементами m_{ij}, определяемыми по формулам

$$m_{ij} = \max \left| \frac{\partial \varphi_i}{\partial x_j} \right|.$$

Достаточным условием сжимаемости отображения φ является условие

$$\|M\| < 1$$

в какой-либо матричной норме. В различных (но эквивалентных) нормах это условие выглядит так

$$\sum_{j=1}^{n} m_{ij} < 1$$

или

$$\sum_{i=1}^{n} m_{ij} < 1$$

или

$$\sum_{i,j=1}^{n} m_{ij}^2 < 1.$$

Ключевые термины

Корень функции — точка, в которой функция принимает нулевое значение.

Метод Ньютона — основной итерационный метод для нахождения приближенного решения системы нелинейных уравнений.

Начальное приближение — начальное значение последовательности, которая рассчитывается с помощью рекуррентной процедуры.

Метод секущих — двухшаговый итерационный метод решения нелинейных уравнений без необходимости вычислять производную.

Краткие итоги: Рассмотрены численные методы нахождения приближенных корней трансцендентных уравнений. Приведена объектно-ориентированная реализация наиболее популярных методов нахождения корней уравнений.

Глава VI

Аппроксимация функций

14. Приближение многочленами

Довольно часто возникает задача восстановление значений функции, которая задана лишь в некоторых точках. Совершенно очевидно, что если мы знаем значения функции лишь в некоторых фиксированных точках, то значения функции в промежуточных значениях аргумента могут быть любыми. Однако часто имеется некоторая априорная информация о свойствах функции, с помощью которой удается найти приемлемое значение функции.

Пусть на отрезке $[a, b]$ задана некоторая числовая функция $f(x)$. **Разбиением отрезка** $[a, b]$ называется конечное множество точек $\{x_n\}_{n=0}^N$ таких, что

$$0 = x_0 < x_1 < \cdots < x_N = b.$$

Точки x_n мы будем называть **узловыми точками**. Пусть также на ряду с разбиением отрезка нам дан набор чисел $\{f_n\}_{n=0}^N$, который имеет смысл значений функции в узловых точках

$$f(x_n) = f_n, \quad n = 0, \ldots, N.$$

Задача **интерполяции** состоит в том, чтобы найти значение функции f в произвольной точке $x \in [a, b]$. Несколько реже возникает за-

дача **экстраполяции** состоящей в том, чтобы найти значение функции f в точке $x \notin [a, b]$. Мы будем в основном рассматривать задачу интерполяции.

Решение задачи интерполяции может быть представлено в виде функции (алгоритма)

$$R(x, x_0, \ldots, x_N, f_0, \ldots, f_N) : \mathbb{R} \times \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \to \mathbb{R}.$$

Однако чаще под решением задачи интерполяции понимают построение такой функции $\tilde{f}_N(x)$, которая определена на всем отрезке $[a, b]$. Эта функция называется **интерполяционной**.

Классическим методом построения интерполяционной функции является построение многочлена степени N

$$P_N(x) = a_N x^N + a_{N-1} x^{N-1} + \cdots + a_1 x + a_0$$

такого, что

$$P_N(x_n) = f_n, \quad n = 0, 1, \ldots, N.$$

Очевидно, что для определения этого многочлена необходимо и достаточно найти значения $\{a_n\}_{n=0}^N$.

Покажем, что такой многочлен всегда можно построить. Точнее мы предъявим формулу, которая даст нам этот интерполяционный многочлен.

Введем функции:

$$p_N^i(x) = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_N)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_N)}.$$

Видно, что эти функции сами являются многочленами порядка N. Непосредственно из этой формулы мы имеем следующие соотношения

$$p_N^i(x_j) = 0, \quad j \neq i$$

и

$$p_N^i(x_i) = 1.$$

Тогда искомый интерполяционный многочлен может быть найден по формуле

$$P_N(x) = \sum_{i=0}^{N} p_N^i(x) f_i. \qquad \text{(VI.1)}$$

Можно убедиться также в том, что этот многочлен единственный. Действительно, для любого другого многочлена, для которого

$$Q_N(x_i) = f_i, \quad i = 0, \ldots, N$$

мы имеем, что

$$Q_N(x_i) = P_N(x_i), \quad i = 0, \ldots, N.$$

Тогда многочлен $L_N(x) = P_N(x) - Q_N(x)$ имеет степень не большую, чем N, а также имеет $N + 1$ корней. По основной теореме алгебры этот многочлен $L_N(x) \equiv 0$.

Многочлен, заданный по формуле (VI.1) называется интерполяционным многочленом в форме Лагранжа.

Лагранжева форма интерполяционного многочлена является не единственной формой интерполяционного многочлена. Более удобной для практических расчетов является интерполяционный многочлен в форме Ньютона. Для заданного разбиения отрезка и значений функции в этих узлах введем понятие раздельной разности. Раздельной разностью первого порядка называется число

$$f(x_0; x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

Раздельная разность N-го порядка определяется по рекуррентной формуле

$$f(x_0; x_1; \ldots; x_N) = \frac{f(x_1; x_1; \ldots; x_N) - f(x_0; x_1; \ldots; x_{N-1})}{x_N - x_0}.$$

Теперь мы можем определить интерполяционный многочлен в форме Ньютона по формуле

$$Q_N(x) = f_0 + f(x_0; x_1)(x - x_0) + \ldots$$

$$+f(x_0; x_1; \ldots; x_N)(x - x_0)(x - x_1)\ldots(x - x_{N-1}).$$

Разумеется, интерполяционный многочлен в форме Ньютона совпадает с интерполяционным многочленом в форме Лагранжа.

Хотя формулы для построения интерполяционных многочленов выглядят просто, метод интерполяции функций многочленами имеет серьезные недостатки для больших значений N. Во-первых, работа с многочленами большой степени, как правило, сопряжено с вычислительной неустойчивостью. Во-вторых, как было показано К. Рунге в своей знаменитой работе 1901 года, существует такая бесконечно гладкая функция, для которой интерполяционный многочлен, построенный на равномерной сетке может иметь бесконечно большое отклонение с увеличением количества узловых точек.

Рассмотрим этот пример. Пусть функция

$$f(x) = \frac{1}{1 + 25x^2},$$

заданная на $[-1, 1]$. Это функция обладает почти всеми «хорошими» свойствами. Для каждого N введем узловые точки по формуле

$$x_i = -1 + i\frac{2}{N}, \quad i = 0, 1, \ldots, N.$$

Через $P_N(x)$ введем интерполяционный многочлен, построенный по формуле (VI.1). К. Рунге было показано, что для этого случая имеет место

$$\lim_{N\to\infty} \max_{x\in[-1,1]} |f(x) - P_N(x)| = \infty.$$

Ключевые термины

Разбиение отрезка — конечное множество точек, принадлежащих заданному отрезку.

Узловые точки — элементы из множества разбиения отрезка.

Интерполяция — процедура приближенного вычисления значений функции по заданным значениям функции в узловых точках.

Экстраполяция — процедура вычисления значений функции вне отрезка, на котором задана функция.

Интерполяционная функция — функция, которая реализует интерполяцию.

Краткие итоги: Рассмотрены методы интерполяции функции на основе приближения полиномами. Приведены методы построения интерполяционных полиномов в форме Лагранжа и в форме Ньютона.

15. Приближение сплайнами

В предыдущей лекции мы рассматривали приближение функций с помощью многочленов. Многочлены являются бесконечно гладкими функциями. С одной стороны это может показаться достоинством интерполяционного метода. Однако на практике это достоинство часто оборачивается недостатком. Налагая на интерполяционную функцию столь большие ограничения, мы неизбежно сталкиваемся с проблемой устойчивости. Чаще гораздо более выгодно использовать в качестве интерполирующих функций меньшие требования гладкости, но получать более эффективные численные решения. Тем более, что во многих задачах, где возникают вопросы интерполяции, требования бесконечной гладкости являются неестественными.

Наиболее успешными средствами интерполяции числовых функций являются методы, основанные на сплайн-интерполяции. Термин «сплайн» происходит от названия гибкой рейки, с помощью которой проводят гладкие кривые через заданные точки. В заданных точках сплайн закреплялся физически. В математике под **сплайном** понимают интерполирующую функцию, производные которой могут иметь разрывы в узловых точках. Классическим вариантом сплайна является **кубические сплайны**. Эти сплайны иногда называют сплайнами Шёнберга по имени математика, который ввел эти сплай-

ны на современном уровне. Кубические сплайны представляют собой дважды непрерывно дифференцируемые функции, которые на отрезка между соседними узлами являются кубическими многочленами.

Пусть также дано разбиение отрезка $[a, b]$, и значения функции в узлах

$$0 = x_0 < x_1 < \cdots < x_N = b,$$

$$f(x_n) = f_n, \quad n = 0, \ldots, N.$$

Кубическим сплайном называется такая функция $S_N(x)$, $x \in [a, b]$, что для этой функции выполнены следующие условия:

1. $S_N(x_n) = f_n$, $n = 0, 1, \ldots, N$.
2. $S_N(x)$, $S'_N(x)$, $S''_N(x)$ являются непрерывными функциями.
3. На каждом отрезке $[x_{n-1}, x_n]$, $n = 1, \ldots, N$ функция S_N представима в виде

$$S_N(x) = a_3^n x^3 + a_2^n x^2 + a_1^n x + a_0^n, \quad x \in [x_{n-1}, x_n], n = 1, \ldots, N.$$

Как мы увидим в дальнейшем — для однозначного задания сплайна нам необходимо дополнительно добавить краевые условия на искомый сплайн. Мы будем рассматривать наиболее характерные условия:

1. $S'_N(x_0) = m_0$, $S'_N(x_N) = m_N$.
2. $S''_N(x_0) = M_0$, $S''_N(x_N) = M_N$.

Перейдем к вопросу построения сплайна. Разумеется для того, чтобы задать сплайн необходимо вычислить набор коэффициентов a_i^N, $i = 0, 1, 2, 3$; $n = 1, 2, \ldots, N$. Однако этот путь является неэффективным. Дело в том, что эти коэффициенты не являются независимыми — на них наложены условия, чтобы обеспечить непрерывность

и непрерывную дифференцируемость первых и вторых производных. Мы будем использовать для задания сплайнов другой подход.

Введем обозначение

$$S'_N(x_n) = m_n, \quad n = 0, 1, \ldots, N.$$

Запишем сплайн S_N в следующей форме

$$S_N(x) = f_n(1-t)^2(1+2t)+f_{n+1}t^2(3-2t)+m_n h_n t(1-t)^2-m_{n+1}h_n t^2(1-t),$$

где $t = \frac{(x-x_n)}{h_n}$, $h_n = x_{n+1} - x_n$.

Условия непрерывности второй производной сплайна определяют систему линейных уравнений

$$\lambda_n m_{n-1} + 2m_n + \mu_n m_{n+1} = 3\left(\mu_n \frac{f_{n+1} - f_n}{h_n} + \lambda_n \frac{f_n - f_{n-1}}{h_{n-1}}\right), \quad \text{(VI.2)}$$

где

$$\mu_n = \frac{h_{n-1}}{h_{n-1} + h_n}, \quad \lambda_n = 1 - \mu_n.$$

К этим условиям следует добавить краевые условия. В итоге для мы имеем следующую систему уравнений

$$2m_0 + \mu_0 m_1 = d_0$$

$$\lambda_n m_{n-1} + 2m_n + \mu_n m_{n+1} = d_n, \ n = 1, \ldots, N-1 \qquad \text{(VI.3)}$$

$$\lambda_N m_{N-1} + 2m_N = d_N$$

где

$$d_n = 3\left(\mu_n \frac{f_{n+1} - f_n}{h_n} + \lambda_n \frac{f_n - f_{n-1}}{h_{n-1}}\right), \ n = 1, \ldots, N-1.$$

Для условий первого типа

$$\mu_0 = \lambda_N = 0, \quad d_0 = 2m_0, \quad d_N = 2m_N,$$

а для условий второго типа

$$\mu_0 = \lambda_N = 1, \quad c_0 = 3\frac{f_1 - f_0}{h_0} - \frac{h_0}{2}M_0, \ c_N = 3\frac{f_N - f_{N_1}}{h_{N-1}} + \frac{h_{N-1}}{2}M_N,$$

Как мы уже отмечали, система (VI.3) представляет собой систему линейных алгебраических уравнений. Однако матрица, определяющая эту систему является трехдиагональной. Для решения таких уравнений будем применять уже известный нам метод прогонки.

Приведем реализацию нашего класса

```
class TSpline
{
    double[] Xn;
    int N;
    TRFunc f;

    TProgon Progon;

    double[] m;

    public TSpline(double[] Xn, TRFunc f)
    {
        this.N = Xn.Count();
        this.Xn = Xn;
        this.f = f;
    }

    double h(int n)
    {
        return Xn[n+1] - Xn[n];
    }

    double mu(int n)
    {
        return h(n - 1) / (h(n - 1) + h(n));
```

```
}

double la(int n)
{
    return 1 - mu(n);
}

public void Build(double m0, double mN)
{
    Progon = new TProgon();
    Progon.a = new double[N];
    Progon.b = new double[N];
    Progon.c = new double[N];
    Progon.d = new double[N];

    int n;

    for (n = 1; n < N-1; n++)
    {
        Progon.a[n] = 2.0;
        Progon.b[n] = la(n);
        Progon.c[n] = mu(n);
        Progon.d[n] = 3.0 * (mu(n) * (f.CalcY(Xn[n + 1])
        - f.CalcY(Xn[n])) / h(n) +
            la(n) * (f.CalcY(Xn[n]) -
            f.CalcY(Xn[n - 1])) / h(n - 1));
    }

    Progon.a[0] = 2;
    Progon.a[N-1] = 2;
```

```
    Progon.b[0] = 0;
    Progon.b[N-1] = 0;
    Progon.c[0] = 0;
    Progon.c[N-1] = 0;

    Progon.d[0] = 2 * m0;
    Progon.d[N-1] = 2 * mN;

    Progon.CalcZ();
}

public double CalcSpline(double x)
{
    int n;

    for (n = 0; n < N-2; n++)
    {
        if ((x >= Xn[n]) && (x < Xn[n + 1]))
        {
            break;
        }
    }

    double t = (x - Xn[n]) / h(n);

    return f.CalcY(Xn[n]) * (1 - t) * (1 - t) * (1 + 2 * t) +
        f.CalcY(Xn[n + 1]) * t * t * (3 - 2 * t) +
        Progon.z[n] * h(n) * t * (1 - t) * (1 - t) -
        Progon.z[n + 1] * h(n) * t * t * (1 - t);
}
```

```
}
```

Теперь испытаем наш сплайн на функции $\sin x$. Мы будем строить сплайн всего по шести точкам.

```
TSinFunc F = new TSinFunc(0, 2 * Math.PI);

int n;
int N = 5;

double[] Xn = new double[N + 1];

double dx = (F.Get_b() - F.Get_a()) / (double)N;

for (n = 0; n <= N; n++)
{
    Xn[n] = F.Get_a() + (double)n * dx;
}

TSpline Spline = new TSpline(Xn, F);

Spline.Build(1, 1);

StreamWriter Fout = File.CreateText("spline.txt");

double x = F.Get_a();

dx = (F.Get_b() - F.Get_a()) / (10.0 * (double)N);

while (x <= F.Get_b())
{
    Fout.WriteLine("{0}\t{1}\t{2}\t{3}", x, F.CalcY(x),
```

```
    Spline.CalcSpline(x), F.CalcY(x) - Spline.CalcSpline(x));

    x += dx;
}

Fout.Close();
```

На рисунке 15.1 мы приводим погрешность нашей интерполяции. Наибольшая погрешность не превосходит 0.01, что оказывается очень неплохим результатом.

Ключевые термины

Сплайн — интерполяционная функция, производные которой могут иметь разрывы в узловых точках.

Кубические сплайны — сплайны, которые являются кубическими многочленами между соседними узловыми точками и являются дважды непрерывно дифференцируемыми.

Краткие итоги: Рассмотрены методы интерполяции функций на основе кубических сплайнов. Приведена объектно-ориентированная реализация построение кубического сплайна.

Глава VII

Дифференциальные уравнения

16. Обыкновенные дифференциальные уравнения

В настоящей лекции мы будем рассматривать задачу Коши для довольного широкого класса обыкновенных дифференциальных уравнений.

Сначала перейдем к формальному определению постановки **задачи Коши**. Пусть $D \subset \mathbb{R}^n$ — область (ограниченная или нет) в n-мерном пространстве. Пусть также задан отрезок $[0, \mathbb{T}]$, здесь $\mathbb{T} > 0$ может быть и бесконечностью. Введем обозначение

$$D_{\mathbb{T}} = \{(x, t) : x \in \overline{D}, t \in [0, \mathbb{T}]\}.$$

Пусть в $D_{\mathbb{T}}$ задана функция $f(x, t)$ тогда можно рассматривать обыкновенное дифференциальное уравнение

$$y'(t) = f(y(t), t), \ t \in [0, \mathbb{T}]. \tag{VII.1}$$

С этим уравнением связывается начальное условие

$$y(0) = y^0, \ y_0 \in D. \tag{VII.2}$$

Задача нахождения функции $y(t)$ такой, что на некотором интервале $[0,T]$ функция $y(t)$ имеет непрерывную производную при $t \in [0,T]$ имеет место $y(t) \in \overline{D}$, и функция $y(t)$ удовлетворяет уравнению (VII.1) и начальному условию (VII.2).

В теории обыкновенных дифференциальных уравнений доказываются следующие теоремы.

Теорема 16.1. *Пусть функция $f(x,t)$ является непрерывной в $D_{\mathbb{T}}$, тогда существует такое $T > 0$, что на $[0,T]$ существует по крайней мере одно решение задачи* (VII.1)–(VII.2).

В условиях теоремы существуют примеры задач Коши, которые имеют более одного решения. Действительно, если взять $D = \mathbb{R}$, $\mathbb{T} = \infty$, а в качестве функции f

$$f(x,t) = \sqrt{x},$$

то задачка Коши

$$y'(t) = \sqrt{y(t)},$$
$$y(0) = 0$$

имеет бесконечное число решений:

$$y(t) = \begin{cases} 0, & t \le \alpha, \\ \frac{(t-\alpha)^2}{4}, & t \ge \alpha, \end{cases}$$

где $\alpha \ge 0$ — параметр семейства решений.

Чтобы гарантировать единственность решения задачи Коши, необходимо накладывать большие условия на функцию f.

Теорема 16.2. *Пусть функция $f(x,t)$ является непрерывной в $D_{\mathbb{T}}$, и существует какая положительная константа $L > 0$, зависящая только от $\tau > 0$ что для любого отрезка $[0,\tau] \subset [0,\mathbb{T}]$ выполнено неравенство*

$$|f(x',t) - f(x'',t)| \le L|x' - x''|, \qquad \text{(VII.3)}$$

для всех $x', x'' \in D$ *и* $t \in [0, \tau]$.

Тогда существует такое $T > 0$, *что на* $[0, T]$ *существует единственное решение задачи* (VII.1)–(VII.2).

Условие (VII.3) называется **условием Липшица**. Это условие будет заведомо выполнено, если непрерывная функция f имеет ограниченные и непрерывные производные по x в D.

Приведенные выше теоремы гарантируют только локальную разрешимость задачи Коши. Действительно, следующий пример задачи Коши не имеет решения на любом отрезке $[0, T]$, где $T \geq \frac{\pi}{2}$. Пусть снова $D = \mathbb{R}^n$, функция $f(x, t)$ имеет вид

$$f(x, t) = x^2 + 1,$$

тогда задача Коши

$$y'(t) = y^2(t) + 1$$

$$y(0) = 0,$$

имеет единственное решение

$$y(t) = \operatorname{tg} t,$$

которое не продолжается вне интервала $[0, \frac{\pi}{2})$.

Чтобы гарантировать существование **глобального по времени решения**, необходимо накладывать дополнительные условия на рост правой части.

Теорема 16.3. *Пусть выполнены все условия теоремы 16.2 и дополнительно функция* $f(x, t)$ *удовлетворяет условию в области* $D = \mathbb{R}^n$

$$|f(x, t)| \leq M(|x| + 1),$$

где константа $M > 0$ *не зависит ни от* x, *ни от* t, *тогда задача Коши имеет единственное решение на отрезке* $[0, \infty)$.

В курсах по обыкновенным дифференциальным уравнениям проходят разнообразные методы нахождения точных решений дифференциальных уравнений. Однако, во-первых, случаи, когда можно найти точные решения являются исключительными случаями и для многих уравнений доказано, что найти решение в квадратурах невозможно. Во-вторых, даже если удалось найти точное решение, то это решение может быть выражено в неявном виде, либо в виде квадратур, которые в свою очередь могут не выражаться в элементарных случаях. В-третьих, во многих задачах правые части могут не быть заданы элементарными функциями. Наконец, во многих случаях построение приближенного решения с достаточной точностью является значительно более простым и эффективным, чем нахождение точного решения.

Существует довольно много численных методов решения задачи Коши для дифференциальных уравнений. При этом многие методы были созданы еще в «до машинную эпоху». Такие методы часто ориентированы на ручной расчет и включают в себя использование аналитических выкладок, например, расчет производных от правой части. Наиболее часто используемым является метод Рунге-Кутта. Этот метод имеет весьма высокую точность, может иметь переменный шаг и может быть легко запрограммирован. Одним из самых простейших методов решения дифференциального уравнения является метод Эйлера. Мы подробно рассмотрим метод Эйлера и метод Рунге-Кутта 4-го порядка.

Метод Эйлера применялся еще Л. Эйлером для доказательства существования решения задачи Коши. Он имеет интуитивно понятную форму, легко может быть запрограммирован, но имеет низкую точность. И так, рассмотрим уравнение (VII.1). Производную в этом уравнении приближенно представим с помощью конечной разности:

$$y'(t) \approx \frac{y(t+h) - y(t)}{h},$$

где $h > 0$ некоторое число. Используя это представление мы вместо дифференциального уравнения (VII.1) получим разностное уравнение

$$\frac{y(t+h) - y(t)}{h} = f(y(t), t).$$

От сюда найдем

$$y(t+h) = y(t) + hf(y(t), t). \tag{VII.4}$$

Пусть мы хотим найти приближенное решение на отрезке $[0, T]$. Разобьем этот отрезок конечным числом точек необязательно равномерно

$$0 = t_0 < t_1 < \cdots < t_N = T.$$

Введем обозначения $h_i = t_i - t_{i-1}$, $i = 1, 2, \ldots, N$. Для простоты будем предполагать, что функция $f(x, t)$ определена во всем пространстве $\mathbb{R}^{n+1}$. Тогда мы можем построить набор $\{y_i\}_{i=0}^N \subset \mathbb{R}^n$ по следующему правилу

$$y_0 = y^0,$$

$$y_i = y_{i-1} + h_i f(y_{i-1}, t_{i-1}),\ i = 1, 2, \ldots, N.$$

Обозначим $h = \max\{h_i : i = 1, 2, \ldots, N\}$. Вектора y_i имеют смысл значений приближенного решения в точках t_i. Для нахождения приближенного решения внутри интервалов (t_{i-1}, t_i) можно применять различные методы интерполяции, например, линейной. Функция $y^N(t)$, построенная с помощью линейной интерполяции, называется ломанной Эйлера. Если для рассматриваемой задачи выполнены условия теоремы 16.2 и на отрезке $[0, T]$ существует решение $y(t)$, то приближенное решение $y^N(t)$ сходится к точному решению равномерно имеет место оценка

$$\max_{t \in [0,T]} |y(t) - y^N(t)| \leq Ch.$$

Таким образов метод Эйлера имеет первый порядок точности.

Реализуем этот метод на C# . Сначала реализуем абстрактный класс, который будет использован для конструирования различных методов построения численных решений систем обыкновенных дифференциальных уравнений.

```
abstract class TODE
{
    public int N;
    protected double t; // текущее время
    // искомое решение Y[0] - само решение,
    // Y[i] - i-тая производная решения
    public double[] Y;
    protected double[] YY; // внутренние переменные

    public TODE(int N) // N - размерность системы
    {
        this.N = N;
        Y = new double[N]; // создать вектор решения
        YY = new double[N]; // и внутренних решений
    }

    // установить начальные условия.
    // t0 - начальное время, Y0 - начальное условие
    public void SetInit(double t0, double[] Y0)
    {
        t = t0;
        int i;
        for (i = 0; i < N; i++)
        {
            Y[i] = Y0[i];
        }
```

```
    }

    public double GetCurrent() // вернуть текущее время
    {
        return t;
    }

    abstract public void F(double t, double[] Y,
    ref double[] FY); // правые части системы.

    // следующий шаг, dt - шаг по времени
    abstract public void NextStep(double dt);
}
```

На основе этого класса построим класс, реализующий метод Эйлера.

```
abstract class TEuler : TODE
{
    public TEuler(int N) : base(N) {}
    public override void NextStep(double dt)
    {
        int i;

        F(t, Y, ref YY);

        for (i = 0; i < N; i++)
        {
            Y[i] = Y[i] + dt * YY[i];
        }

        t = t + dt;
    }
```

```
}
```

Прежде чем испытать наш метод Эйлера, мы рассмотрим и реализуем метод Рунге-Кутта 4-го порядка. Метод Рунге-Кутта, также как и метод Эйлера, допускает перемену шага, но имеет значительно большую точность. Пусть t_i и y_i имеют тот же смысл, что и при рассмотрении метода Эйлера.

Правила построения точек y_i следующие

$$y_i = y_{i-1} + \frac{h_i}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

где

$$k_1 = f(y_{i-1}, t_{i-1}),$$
$$k_2 = f\left(y_{i-1} + \frac{h}{2}k_1, t_{i-1} + \frac{h}{2}\right),$$
$$k_3 = f\left(y_{i-1} + \frac{h}{2}k_2, t_{i-1} + \frac{h}{2}\right),$$
$$k_4 = f(y_{i-1} + hk_3, t_{i-1} + h).$$

Как мы видим, для расчета решения на следующем шаге необходимо вычислить правую часть четыре раза, зато точность этого метода имеет четвертый порядок, при условии гладкости правой части.

Перейдем к реализации метода Рунге-Кутта на основе нашего класса $TODE$.

```
abstract class TRungeKutta : TODE
{

    double[] Y1, Y2, Y3, Y4; // внутренние переменные

    public TRungeKutta(int N) : base(N)
    {
        Y1 = new double[N];
```

```
            Y2 = new double[N];
            Y3 = new double[N];
            Y4 = new double[N];
        }

// следующий шаг метода Рунге-Кутта, dt - шаг по времени
        override public void NextStep(double dt)
        {
            if (dt < 0)
            {
                return;
            }

            int i;

            F(t, Y, ref Y1); // рассчитать Y1

            for (i = 0; i < N; i++)
            {
                YY[i] = Y[i] + Y1[i] * (dt / 2.0);
            }
            F(t + dt / 2.0, YY, ref Y2); // рассчитать Y2

            for (i = 0; i < N; i++)
            {
                YY[i] = Y[i] + Y2[i] * (dt / 2.0);
            }
            F(t + dt / 2.0, YY, ref Y3); // рассчитать Y3

            for (i = 0; i < N; i++)
            {
```

```
            YY[i] = Y[i] + Y3[i] * dt;
        }
        F(t + dt, YY, ref Y4); // рассчитать Y4

        for (i = 0; i < N; i++)
        {
            // рассчитать решение на новом шаге
            Y[i] = Y[i] + dt / 6.0 *
            (Y1[i] + 2.0 * Y2[i] + 2.0 * Y3[i] + Y4[i]);
        }

        t = t + dt; // увеличить шаг

    }
}
```

Теперь протестируем наши методы на примере задач Коши, описывающей гармонические колебания математического маятника. Мы решаем следующую задачу.

$$y''(x) + y(x) = 0$$

с начальными условиями

$$y(0) = 0,$$

$$y'(0) = 1.$$

Эта задача имеет единственное решение — $\sin t$. Чтобы применить к этой задаче наши методы нужно записать ее в виде системы уравнений первого порядка.

$$Y_0(t) = y(t),$$

$$Y_1(t) = y'(t).$$

После такой замены мы имеет следующую систему

$$Y_0'(t) = Y_1(t)$$

$$Y_1'(t) = -Y_0(t)$$

с начальными условиями

$$Y_0(0) = 0,$$

$$Y_1(0) = 1.$$

Реализуем для нашей системы методы Эйлера и Рунге-Кутты

```
class THarmonicEuler : TEuler
{
    public THarmonicEuler() : base(2) { }
    public override void F(double t, double[] Y,
    ref double[] FY)
    {
        FY[0] = Y[1];
        FY[1] = -Y[0];
    }
}

class THarmonicRK : TRungeKutta
{
    public THarmonicRK() : base(2) { }
    public override void F(double t, double[] Y,
    ref double[] FY)
    {
        FY[0] = Y[1];
        FY[1] = -Y[0];
    }
}
```

Испытаем наши классы

```
double h = 0.1;

double[] Y0 = { 0, 1.0 };

THarmonicEuler HarmonicEuler = new THarmonicEuler();
HarmonicEuler.SetInit(0, Y0);

THarmonicRK HarmonicRK = new THarmonicRK();
HarmonicRK.SetInit(0, Y0);

StreamWriter F = File.CreateText("harmonic.txt");

double t, Euler, RK, Sin;

while (HarmonicEuler.GetCurrent() < (2 * Math.PI + h / 2.0))
{
    t = HarmonicEuler.GetCurrent();
    Euler = HarmonicEuler.Y[0];
    RK = HarmonicRK.Y[0];
    Sin = Math.Sin(t);

    F.WriteLine("{0}\t{1}\t{2}\t{3}\t\t{4}\t{5}\t{6}",
    t, Euler, RK, Sin, t, Math.Abs(Sin - Euler),
    Math.Abs(Sin - RK));

    HarmonicEuler.NextStep(h);
    HarmonicRK.NextStep(h);
}
F.Close();
```

Результаты расчетов приведем на трех графиках. На рисунке 16.1 мы приводим график точного решения и приближенных, полученных методами Эйлера и Рунге-Кутты. Однако на этом графике не возможно отличить точное решение от приближенного решения, полученного методом Рунге-Кутты. На рисунках 16.2 и 16.3 мы показываем погрешности методов. Из анализа этих графиков видно, что точность метода Рунге-Кутты значительно выше, что обосновывается теоретически.

Ключевые термины

Глобальное решение — решение задачи Коши, существующее при всех $t \geq 0$.

Задача Коши — дифференциальное уравнение с заданными начальными условиями на решение.

Метод Рунге-Кутта — наиболее распространенный метод численного интегрирования задачи Коши с точностью четвертого порядка.

Метод Эйлера — простейший метод численного интегрирования задачи Коши с точностью первого порядка.

Условие Липшица — условие на приращение функции, более сильное чем условие непрерывности, но слабее чем условие дифференцируемости.

Краткие итоги: Построены объектно-ориентированные средства для численного решения задачи Коши. Проведены вычислительные эксперименты, для сравнения методов Эйлера и Рунге-Кутты.

17. Эволюционные уравнения в частных производных

В предыдущей лекции мы рассматривали системы обыкновенных дифференциальных уравнений конечной размерности. Однако многие процессы в нашем мире описываются бесконечными система-

ми дифференциальных уравнений. Такие системы иногда называют **распределенными системами** или уравнениями в частных производных. Решением уравнений в частных производных является функция многих переменных. Простейшими уравнениями в частных производных второго порядка являются следующие уравнения

$$\Delta u(x) = f(x) \quad \text{— уравнение Лапласа}$$

$$u_t(t,x) = \Delta u(t,x) + f(t,x) \quad \text{— уравнение теплопроводности}$$

$$u_{tt}(t,x) = \Delta u(t,x) + f(t,x) \quad \text{— волновое уравнение}$$

Уравнение Лапласа является примером эллиптического уравнения, уравнение теплопроводности является примером параболического уравнения, а волновое уравнение является примером гиперболического уравнения. С математической точки зрения дифференциальные уравнения в частных производных представляют собой весьма сложную тему, которая имеет принципиальные отличия от обыкновенных дифференциальных уравнений. С вычислительной точки зрения, часто, дифференциальные уравнения в частных производных могут быть аппроксимированы конечномерными уравнениями. Эллиптические уравнения могут быть аппроксимированы системами линейных алгебраических уравнений, а эволюционные уравнения аппроксимируются системами обыкновенных дифференциальных уравнений.

Мы будем рассматривать эволюционные уравнения относительно функций заданных на отрезке $[0,T]$ со значениями в банаховом пространстве, например, в пространстве $C[a,b]$. Численные методы, которые мы будем рассматривать могут быть одинаково эффективно применены как для линейных, так и для нелинейных уравнений. Рассмотрим математическую постановку начальной задачи. Пусть X есть некоторое банахово пространство. Пусть в этом в этом пространстве задано множество $\mathcal{D} \subset X$, на котором определен оператор $\mathcal{A}$ (линейный или нелинейный).

Будем рассматривать уравнение

$$u_t(t) = \mathcal{A}u(t), \quad t \in [0, T] \tag{VII.5}$$

с начальным условием

$$u(0) = \varphi. \tag{VII.6}$$

Искомой функцией в задаче (VII.5)–(VII.6) является функция $u \in C^1([0, T]; X)$ такая, что $u(t) \in \mathcal{D}$ при всех $t \in [0, T]$ и, удовлетворяющая (VII.5)–(VII.6). Элемент $\varphi \in \mathcal{D}$ называется **начальным условием**, а задача (VII.5)–(VII.6) называется **абстрактной задачей Коши**.

Приведем два характерных примера таких задач. В качестве пространства X мы возьмем пространство непрерывных функций $C[0, 2\pi]$ в качестве множества $\mathcal{D}$ возьмем множество непрерывно дифференцируемых функций на отрезке $[0, 2\pi]$ и являющихся 2π-периодическими. Будем рассматривать два модельных уравнения. Первое — уравнение линейного переноса

$$u_t(t, x) = cu_x(t, x), \tag{VII.7}$$

где $c \neq 0$ — постоянная, имеющая смысл «скорости» распространения волны. Второе уравнение — уравнение нелинейного переноса

$$u_t(t, x) = u(t, x)u_x(t, x). \tag{VII.8}$$

Опишем построение численной схемы. Мы будем использовать **проекционный метод** для приближенного решения абстрактной задачи Коши. Этот метод позволяет свести задачу к системе обыкновенных дифференциальных уравнений. Пусть для любого $n > 0$ существует пара отображений

$$P_N : X \to \mathbb{R}^n, \quad I_N : \mathbb{R}^n \to \mathcal{D}.$$

Как правило на эти отображения накладываются условия

$$P_n I_n x = x, \quad \text{для любого } x \in \mathbb{R}^n$$

и

$$\lim_{n\to\infty} I_N P_N x = x, \quad \text{для любого } x \in \mathcal{D},$$

где предел понимается в метрике пространства X. Задача (VII.5)–(VII.6) заменяется следующей задачей

$$u_t^n(t) = P_n \mathcal{A} I_n u^n(t), \tag{VII.9}$$

$$u_t^n(0) = P_n v f. \tag{VII.10}$$

Задача (VII.9)–(VII.10) представляет собой задачу Коши для системы обыкновенных дифференциальных уравнений n-го порядка. Далее эта система решается стандартными численными методами, например, методом Рунге-Кутта, который мы рассматривали на прошлой лекции. После нахождения решения задачи (VII.9)–(VII.10) то есть функции $u^n(t)$, в качестве приближенным решением исходной задачи можно выбрать функцию $I_n u^n(t)$.

Хотя описанный проекционный метод является, как правило, легко реализуемым, но при его использовании необходимо иметь в виду вопросы, связанные с его сходимостью и устойчивостью. Дело в том, что даже для простейших уравнений при использовании проекционного метода следует согласовывать шаг по времени, то есть тот шаг, который используется в численном методе при решении задачи (VII.9)–(VII.10), с шагом, который имеет место при построении аппроксимации пространства X.

Для уравнений (VII.7) и (VII.8) мы будем использовать следующие операторы P_n и I_n

$$P_n f(x) = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$$

где $f_i = f((i-1)\frac{2\pi}{n})$, $i = 1, 2, \ldots, n$. Для реализации оператора I_n необходимо использовать подходящую интерполяцию. Мы будем рас-

сматривать кусочно-линейную интерполяцию. Покажем, как таким образом можно определить операцию

$$P_n \frac{d}{dx} I_n f = g = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}$$

где

$$g_i = \frac{f_{i+1} - f_i}{h}, \quad i = 1, 2 \ldots, n-1,$$

$$g_n = \frac{f_1 - f_n}{h},$$

где $h = \frac{2\pi}{n}$.

Реализуем этот метод для линейного уравнения переноса. Для этого мы создадим класс, являющийся наследником от класса $TRungeKutta$.

```
class TCUEvol : TRungeKutta
{
    double c; // скорость волны
    double h;

    public TCUEvol(int N, double c) : base(N)
    {
        this.c = c;
        h = 2.0 * Math.PI / N;
    }

    public override void F(double t, double[] Y, ref double[] FY)
    {
        int i;
        for (i = 0; i < N - 1; i++)
        {
            FY[i] = c * (Y[i + 1] - Y[i]) / h;
```

```
        }
        FY[N - 1] = c * (Y[0] - Y[N - 1]) / h;
    }
}
```

Испытаем наш класс, учитывая, что для начальной функции вида $\sin kx$ это уравнение имеет точное решение в виде бегущей волны $u(t,x) = sin(k(t+x))$.

```
int N = 1000;
double h = 0.0001;

double[] Y0 = new double[N];

TCUEvol CU = new TCUEvol(N, 1);

double x;
int i;
for (i = 0; i < N; i++)
{
    x = (double)i * (2.0 * Math.PI) / (double)N;
    Y0[i] = Math.Sin(5.0 * x);
}

CU.SetInit(0, Y0);

double t = 0;

while (CU.GetCurrent() < (1.0 + h / 2.0))
{
    t = CU.GetCurrent();
```

```
    CU.NextStep(h); // рассчитать на следующем шаге
}

StreamWriter Fout = File.CreateText("cu.txt");

for (i = 0; i < N; i++)
{
    x = (double)i * (2.0 * Math.PI) / (double)N;
    Fout.WriteLine("{0}\t{1}\t{2}\t{3}", x, CU.Y[i],
    Math.Sin(5.0 * (t + x)), Math.Sin(5.0 * (t + x)) - CU.Y[i]);
}

Fout.Close();
```

Мы решали задачу:

$$u_t(t,x) = u_x(t,x)$$

$$u(0,x) = \sin 5x.$$

Эта задача с 2π-периодическими по переменной x условиями имеет единственное решение

$$u(t,x) = \sin(5(t+x)),$$

в чем можно убедится непосредственно. На рисунке 17.1 мы приводим графики точного решения (сплошной линией) и приближенного (точечной линией) при $t = 1.0$. Результат нашего вычислительного опыта не является удовлетворительным. Мы видим хорошее совпадение фазы решений, но амплитуда приближенного решения меньше точного. Наша схема оказалась весьма диссипативной, что делает ее непригодной. На рисунке 17.2 мы приведем погрешность приближенного решения.

Покажем как можно существенно повысить точность нашего метода. Для этого мы воспользуемся так называемыми **аналитико-**

числовыми методами. Суть этих методов состоит в том, что ряд операций, можно выполнить точно. Например в известной нам уже задаче дифференцирования функции. Пусть нам нужно найти производную функции $f(x) = a \sin \alpha x + b \cos \beta x$. При программировании нам выгодно задавать не поточечные значения этой функции, а всего четыре коэффициента — a, b, α и β. Обозначим эти коэффициенты четырьмя переменными:

```
double sin_a; \\ a

double sin_alpha; \\ alpha

double cos_b; \\ b

double cos_beta; \\ beta
```

Тогда производная функции $f'(x)$ имеет такой же вид и может быть также представлена такими же коэффициентами:

```
double Dsin_a; \\ a

double Dsin_alpha; \\ alpha

double Dcos_b; \\ b

double Dcos_beta; \\ beta
```

Вычислить коэффициенты производной можно следующим образом

```
Dsin_a = -cos_b * cos_beta;

Dsin_alpha = cos_beta;
```

```
Dcos_b = sin_a * sin_alpha;

Dcos_beta = sin_alpha;
```

При это, если не считать возможные ошибки округления при умножении, операция дифференцирования была выполнена безукоризненно.

С другой стороны гладкие (например, кусочно дифференцируемые и имеющие ограниченные производные) периодические функции могут быть разложены в ряд Фурье, а ряд Фурье представляет собой комбинацию рассмотренных выше функций. Поэтому, если задавать функцию коэффициентами Фурье, то такие операции как дифференцирование и интегрирование можно выполнять фантастически быстро. Чем более гладкая функция, тем быстрее коэффициенты Фурье сходятся к нулю. Следовательно, часто ряд Фурье удается аппроксимировать конечными тригонометрическими суммами.

Реализуем новый класс в котором мы будем применять численно-аналитические методы.

```
class TCUAn : TRungeKutta
{
    double c;
    double h;

    public TCUAn(int N, double c)
        : base(N)
    {
        this.c = c;
        h = 2.0 * Math.PI / N;
    }

    public double GetY(double x)
    {
```

```
    double res = 0;
    int i;

    double di;
    for (i = 1; i < (N - 1) / 2; i++)
    {
        di = (double)i;

        res += Math.Sin(di * x) * Y[i - 1];

        res += Math.Cos(di * x) * Y[N - i];
    }

    res += Y[(N - 1) / 2];

    return res;

}

public override void F(double t, double[] Y, ref double[] FY)
{
    Diff(Y, ref FY);
    int i;
    for (i = 0; i < N; i++)
    {
        FY[i] = c * FY[i];
    }
}

void Diff(double[] Y, ref double[] DY)
{
```

```
        int i;
        double di;
        for (i = 1; i < (N-1) / 2; i++)
        {
            di = (double)i;
            DY[i - 1] = -di * Y[N - i];
            DY[N - i] = di * Y[i - 1];
        }
        DY[(N - 1) / 2] = 0;
    }
}
```

В этом классе мы добавили новые методы — для дифференцирования функции, а также для суммирования тригонометрической суммы. Заметим, что мы рассматриваем систему дифференциальных уравнений относительно коэффициентов Фурье. Испытаем наш класс.

```
int N = 257;
double h = 0.001;

double[] Y0 = new double[N];

TCUAn CUAn = new TCUAn(N, 1);

for (i = 0; i < N; i++)
{
    Y0[i] = 0;
}

Y0[4] = 1;

CUAn.SetInit(0, Y0);
```

```
t = 0;
while (CUAn.GetCurrent() < (1.0 + h / 2.0))
{
    t = CUAn.GetCurrent();

    CUAn.NextStep(h);
}

Fout = File.CreateText("cuan.txt");

for (i = 0; i < N; i++)
{
    x = (double)i * (2.0 * Math.PI) / (double)N;
    Fout.WriteLine("{0}\t{1}\t{2}\t{3}", x, CUAn.GetY(x),
    Math.Sin(5.0 * (t + x)), Math.Sin(5.0 * (t + x))
    - CUAn.GetY(x));
}

Fout.Close();
```

На рисунке 17.3 мы приводим погрешность приближенного решения.

Сейчас мы рассмотрим нелинейное уравнения переноса

$$u_t(t,x) = u(t,x)u_x(t,x)$$

также с периодическим краевыми условиями. Для этого уравнения характерен эффект обрушения волны. Действительно, в этом уравнении скорость переноса пропорциональна амплитуде. Поэтому «верхушка» волны движется быстрее основания, и в определенный момент производная решения будет стремится к бесконечности. Мы приведем приближенное решение уравнения нелинейного переноса.

Для этого мы также будем использовать аналико-численные методы. Для этого нам нужно реализовать операцию произведения рядов Фурье, что вполне возможно. Приведем код этого класса.

```
class TUU : TRungeKutta
{
    double h;

    public TUU(int N)
        : base(N)
    {
        h = 2.0 * Math.PI / N;
    }

    public double GetY(double x)
    {
        double res = 0;
        int i;

        double di;
        for (i = 1; i < (N - 1) / 2; i++)
        {
            di = (double)i;

            res += Math.Sin(di * x) * Y[i - 1];
            res += Math.Cos(di * x) * Y[N - i];
        }

        res += Y[(N - 1) / 2];

        return res;
```

```
}

public override void F(double t, double[] Y, ref double[] FY)
{
    double[] DU = new double[N];

    Diff(Y, ref DU);

    Mult(Y, DU, ref FY);
}

int kSin(int k)
{
    return k - 1;
}

int kCos(int k)
{
    return N - k;
}

void Mult(double[] U, double[] DU, ref double[] UDU)
{
    int i;
    for (i = 0; i < N; i++)
    {
        UDU[i] = 0;
    }

    int N2 = (N - 1)/2;
```

```
int k, m;
for (k = 1; k < (N - 1) / 2; k++)
{
    for (m = 1; m < (N - 1) / 2; m++)
    {
        // sin kx * sin mx
        if (k != m)
        {
            UDU[kCos(Math.Abs(k - m))] +=
            0.5 * U[kSin(k)] * DU[kSin(m)];
        }
        if (k + m < N2)
        {
            UDU[kCos(k + m)] +=
            -0.5 * U[kSin(k)] * DU[kSin(m)];
        }

        // sin kx * cos mx
        if (k + m < N2)
        {
            UDU[kSin(k + m)] +=
            0.5 * U[kSin(k)] * DU[kCos(m)];
        }
        if (k > m)
        {
            UDU[kSin(k - m)] +=
            0.5 * U[kSin(k)] * DU[kCos(m)];
        }
        if (k < m)
        {
```

```
    UDU[kSin(m - k)] +=
    -0.5 * U[kSin(k)] * DU[kCos(m)];
}

// cos kx * sin km
if (k + m < N2)
{
    UDU[kSin(k + m)] +=
    0.5 * U[kCos(k)] * DU[kSin(m)];
}
if (m > k)
{
    UDU[kSin(m - k)] +=
    0.5 * U[kCos(k)] * DU[kSin(m)];
}
if (m < k)
{
    UDU[kSin(k - m)] +=
    -0.5 * U[kCos(k)] * DU[kSin(m)];
}

// cos kx * cos mx
if (k + m < N2)
{
    UDU[kCos(k + m)] +=
    0.5 * U[kCos(k)] * DU[kSin(m)];
}
if (k != m)
{
    UDU[kCos(Math.Abs(k - m))] +=
    0.5 * U[kCos(k)] * DU[kSin(m)];
```

```
            }
        }
    }
}

void Diff(double[] Y, ref double[] DY)
{
    int i;
    double di;
    for (i = 1; i < (N - 1) / 2; i++)
    {
        di = (double)i;
        DY[i - 1] = -di * Y[N - i];
        DY[N - i] = di * Y[i - 1];
    }
    DY[(N - 1) / 2] = 0;
}
}
```

Теперь испытаем этот класс.

```
int N = 257;
double h = 0.001;

double[] Y0 = new double[N];

TUU UU = new TUU(N);

for (i = 0; i < N; i++)
{
    Y0[i] = 0;
}
```

```
Y0[1] = 1;

UU.SetInit(0, Y0);

t = 0;
while (UU.GetCurrent() < (0.3 + h / 2.0))
{
    t = UU.GetCurrent();

    UU.NextStep(h);
}

Fout = File.CreateText("uu.txt");

for (i = 0; i < N; i++)
{
    x = (double)i * (2.0 * Math.PI) / (double)N;
    Fout.WriteLine("{0}\t{1}", x, UU.GetY(x));

}

Fout.Close();
```

На графике 17.4 мы приведем график приближенного решения при $t = 0.3$. К сожалению мы не имеем точного решения нелинейного уравнения, поэтому мы не приводим графика погрешности приближенного решения. Видно, что волна «пытается» обрушится.

Ключевые термины

Распределенные системы — бесконечно мерные системы дифференциальных уравнений.

Начальное условие — функция, которой равно решение в начальный момент.

Абстрактная задача Коши — дифференциальное уравнение для функций со значениями в банаховом пространстве с заданным начальным условием.

Проекционный метод — метод аппроксимации эволюционных уравнений в частных производных системами обыкновенных дифференциальных уравнений.

Аналитико-числовые методы — численные методы которые содержат операции, выполняемые с использованием аналитического представления математических объектов.

Краткие итоги: Рассмотрены проекционные методы решения эволюционных уравнений в частных производных. Показано, что использование аналитико-числовых методов позволяет существенно улучшить точность расчетов.

Часть III

Приложения в математическом моделировании

Глава VIII

Объектно-ориентированное моделирование динамических систем

18. Объектно-ориентированное управление решениями дифференциальных уравнений

Многие процессы, описываемые дифференциальными уравнениями, являются управляемыми системами. Дадим формальное описание **управляемой системы**.

Пусть D, $D_{\mathbb{T}}$ имеют тот же смысл, что и в прошлой лекции. Пусть теперь еще выделено семейство множеств $U(x,t) \subset \mathbb{R}^m$, где $x \in \mathbb{R}^n$ и $t \in [0,\mathbb{T}]$ являются параметрами. Множество $U(x,t)$ называется **множеством допустимых управлений**.

Пусть теперь задана функция $f(x,u,t)$, которая определена при $x \in \overline{D}$, $u \in \bigcup\limits_{x\in\overline{D},t\in[0,\mathbb{T}]} U(x,t)$, $t \in [0,\mathbb{T}]$. Будем рассматривать обыкновенное дифференциальное уравнение с управлением

$$\begin{gathered} y'(t) = f(y(t),u(t),t), \\ u(t) \in U(y(t),t)\ t \in [0,\mathbb{T}]. \end{gathered} \tag{VIII.1}$$

и начальным условием

$$y(0) = y^0,\ y_0 \in D. \tag{VIII.2}$$

Решением задачи (VIII.1)–(VIII.2) на $[0, T]$ является измеримая функция $u(t) \in U(y(t), t)$ и абсолютно непрерывная функция $y(t) \in \overline{D}$, $t \in [0, T]$, и эти функции удовлетворяют (VIII.1)–(VIII.2).

Мы сформулировали задачу на управления решениями дифференциальных уравнений в довольно общем виде. Однако, как правило, управление осуществляется с некоторой целью. Для этого вводится так называемый целевой функционал. **Целевым функционалом** называется любая числовая функция определенная на решении задачи (VIII.1)–(VIII.2). Обозначим этот функционал следующим образом

$$I = I(y, u) \in \mathbb{R}.$$

При наличии целевого функционала мы приходим к задаче оптимального управления, то есть задаче о нахождении такого решения, которое доставляет максимум (или минимум) целевому функционалу. На практике, однако, часто применяется понятие решение задачи управления, удовлетворительное с точки зрения целевого функционала. Мы будем говорить, что решение задачи управления — пара функций y и u является удовлетворительным с точки зрения целевого функционала, если

$$I(y, u) \in G,$$

где $G \subset \mathbb{R}$ называется удовлетворительной областью значения целевого функционала. Заметим, что G может быть произвольным непустым множеством.

Обратимся к моделированию задач управления. Общая схема такова

1. Задаем начальное управление

2. Решаем систему дифференциальных уравнений с начальным управлением на одном шаге методом Рунге-Кутты

3. По полученному решению на очередном шаге меняем (в случае необходимости) управление

4. Повторяем шаги 2-3 до тех пор пока решение и управления не станут удовлетворительными с точки зрения целевого функционала

Следует отметить, что далеко не всегда удовлетворительное с точки зрения целевого функционала решение задачи управления является оптимальным решением. Более того, существует такие задачи управления решениями, для которых не существует оптимального решения.

Реализуем эту схему в виде абстрактного класса, который мы создадим, как наследник класса $TRungeKutta$.

```
abstract class TControlSystem : TRungeKutta
{
    public int M;
    public double[] U;
    public TControlSystem(int N, int M) : base(N)
    {
        this.M = M;
        U = new double[M];
    }

    abstract public void SetU();

    public override void NextStep(double dt)
    {
        SetU();
```

```
            base.NextStep(dt);
        }
    }
```

Рассмотрим модельную систему управления на основе уравнений, описывающих математический маятник. Будем рассматривать следующую управляемую систему дифференциальных уравнений.

$$y''(t) + u(t)y'(t) + y(t) = 0,$$

с начальными условиями

$$y(0) = 1,$$

$$y'(0) = 0.$$

Здесь управление осуществляет изменением коэффициента затухания. Мы будем рассматривать множество допустимых управлений состоящее всего из двух точек:

$$U = \{0, 1\} \subset \mathbb{R}.$$

Нулевое значение управления соответствует отсутствию затухания в системе, а равное единице — наличию постоянного затухания. При отсутствии затухания наша система будет совершать гармонические колебания. Предположим, что наша цель состоит в том, чтобы поддерживать гармонические колебания с амплитудой 0.2. Мы будем использовать так называемое программное управление, т.е. будем выбирать наше управление зависящим от текущего положения системы. Будем использовать следующую формулу нашего управления:

$$u(y,t) = \begin{cases} 0, & |y| \leq 0.2 \\ 1, & |y| > 0.2 \end{cases}$$

Таким образом, если положение осциллятора меньше заданного, то мы не используем затухания, а в случае, когда положение осциллятора больше заданного, мы используем затухание. Реализуем сказанное.

```
class TOscControl : TControlSystem
{
    public TOscControl() : base(2, 1)
    {
        U[0] = 0;
    }

    public override void SetU()
    {
        if (Math.Abs(Y[0]) > 0.2)
        {
            U[0] = 1;
        }
        else
        {
            U[0] = 0;
        }
    }

    public override void F(double t, double[] Y,
    ref double[] FY)
    {
        FY[0] = Y[1];
        FY[1] = -Y[0] - U[0] * Y[1];
    }
}
```

И проведем вычислительный эксперимент.

```
TOscControl Osc = new TOscControl();
Osc.SetInit(0, new double[2] { 1, 0 });
```

```
double h = 0.01;

StreamWriter F = File.CreateText("control.txt");
while (Osc.GetCurrent() < 50.0 + h / 2.0)
{
    Console.WriteLine("{0}\t{1}\t{2}\t{3}",
    Osc.GetCurrent(), Osc.Y[0], Osc.Y[1], Osc.U[0]);
    F.WriteLine("{0}\t{1}\t{2}\t{3}",
    Osc.GetCurrent(), Osc.Y[0], Osc.Y[1], Osc.U[0]);

    Osc.NextStep(h);
}
F.Close();
```

На рисунке 18.1 мы приводим фазовый портрет нашей управляемой системы. Угловые точки на траектории отражают моменты переключения управления. Мы видим, что траектория быстро сходится к искомому предельному циклу. Предельным циклом называется периодическая (замкнутая) траектория в окрестности которой нет других периодических траекторий.

Заметим, что хотя наше уравнение маятника является линейным относительно неизвестной функции y и управление входит в это уравнение линейно, но зависимость решения от управления является уже нелинейной. Эта ситуация является общей для систем управления.

Другой пример управляемой системы состоит в следующем. Будем рассматривать простейшую систему автопилота самолета. Мы будем управлять курсовым углом самолета. Пусть нам задан нулевой курс. В линейном приближении колебания этого курса описываются системой дифференциальных уравнений:

$$y''(t) = -y'(t) + u(t)$$

пусть начальные условия таковы

$$y(0) = 1,$$

$$y'(0) = 0.$$

Предположим, что наше управление может принимать только два значения:

$$U = \{-1, 1\} \subset \mathbb{R}.$$

Программное управление выберем следующим естественным образом

$$u(x,t) = \begin{cases} 1, & y < 0 \\ -1, & y > 0 \end{cases}$$

Реализуем эту систему

```
class TPilotControl : TControlSystem
{
    public TPilotControl()
        : base(2, 1)
    {
        U[0] = 1;
    }

    public override void SetU()
    {
        if (Y[0] < 0)
        {
            U[0] = 1;
        }
        else
        {
            U[0] = -1;
        }
```

```
        }

        public override void F(double t, double[] Y,
        ref double[] FY)
        {
            FY[0] = Y[1];
            FY[1] = -Y[1] + U[0];
        }
    }
```

И проведем вычислительный эксперимент с нашим управлением.

```
TPilotControl Pilot = new TPilotControl();
Pilot.SetInit(0, new double[2] { 1, 0 });

F = File.CreateText("pilot.txt");
while (Pilot.GetCurrent() < 50.0 + h / 2.0)
{
    Console.WriteLine("{0}\t{1}\t{2}\t{3}",
    Pilot.GetCurrent(), Pilot.Y[0], Pilot.Y[1], Pilot.U[0]);
    F.WriteLine("{0}\t{1}\t{2}\t{3}",
    Pilot.GetCurrent(), Pilot.Y[0], Pilot.Y[1], Pilot.U[0]);

    Pilot.NextStep(h);
}
F.Close();
```

В результате на рисунке 18.2 мы получим колебания нашего курса. Можно видеть, что довольно быстро амплитуда колебаний нашего курса становится малой, однако в нашей системе возникли характерные автоколебания. Автоколебаниями называются незатухающие периодические колебания. При этом частота этих колебаний определяется свойствами самой нелинейной системы, а не частотой внешнего

воздействия. Предельный цикл является одним из примеров автоколебаний.

Ключевые термины

Множество допустимых управлений — множество, которому принадлежит управление.

Управляемая система — система, описывающаяся дифференциальным и уравнениями, содержащие управление.

Целевой функционал — функционал на решении, с помощью которого определяется оптимальное решение.

Краткие итоги: Рассмотрены методы управления дифференциальными уравнениями с помощью выбора правых частей. Проведены вычислительные эксперименты, позволяющие дать качественную картину некоторых задач управления.

19. Общие динамические системы

Динамической системой называют математическую модель системы, процессы в которой развиваются во времени. Состояние динамической системы характеризуется **фазовыми координатами**, которые принадлежат фазовому пространству. Время в динамической системе может быть как непрерывным, так и дискретным. Фазовое множество может быть конечным или бесконечным.

Пусть дано непустое множество $\mathcal{A}$, которое мы будем называть фазовым пространством, а каждый элемент этого множества $a \in \mathcal{A}$ описывает состояние рассматриваемой системы. Через T мы будем обозначать либо множество $\mathrm{R}_+ = \{t \in \mathbb{R} : t \geq 0\}$, либо множество натуральных чисел $\mathbb{N} = \{0, 1, 2, \dots\}$. Это множество будет играть роль времени. В первом случае мы будем рассматривать непрерывные динамические системы, а во втором — дискретные.

Абстрактно заданной динамической системой называется

однопараметрическая полугруппа преобразований множества $\mathcal{A}$

$$p : \mathcal{A} \times \mathrm{T} \to \mathcal{A},$$

удовлетворяющее следующим условиям:

1. $p(a, 0) = a$ для любого $a \in \mathcal{A}$,
2. $p(p(a, t_1), t_2) = p(a, t_1 + t_2)$ для любых $a \in \mathcal{A}$, $t_1, t_2 \in \mathrm{T}$.

Соответственно, чтобы задать динамическую систему необходимо фиксировать множества $\mathcal{A}$, T и задать отображение p, удовлетворяющее указанным выше условиям. Пусть $\mathcal{A} = \mathbb{R}^n$, $\mathrm{T} = \mathrm{R}_+$. Рассмотрим систему обыкновенных дифференциальных уравнений n-го порядка, которая имеет единственное глобальное решение для всех $y^0 \in \mathbb{R}^n$. Тогда решение этой системы задает отображение

$$y = y(y^0, t) : \mathbb{R}^n \times \mathbb{R}_+ \to \mathbb{R}^n.$$

Легко видеть, что это отображение удовлетворяет всем условиям непрерывной динамической системы.

Дискретную динамическую систему можно построить следующим образом. Пусть у нас некоторое множество $\mathcal{A}$ и задан оператор

$$T : \mathcal{A} \to \mathcal{A}.$$

Построим полугруппу отображений $p(a, n)$ следующим образом

$$p(a, n) = T^n a, \ n \in \mathbb{N},$$

где T^0 есть единичное отображение. Таким образом мы имеем дискретную динамическую систему. Содержательные примеры конечной дискретной динамической системы строятся на основании конечных автоматов.

Конечным автоматом называется набор $\mathbb{M} = \{\Sigma, Q, q_0, \Phi\}$, где Σ — конечное множество (входной/выходной алфавит), Q — конечное

множество (внутренние состояния), $q_0 \in Q$ — начальное состояние, Φ — функция переходов

$$\Phi : \Sigma \times Q \to \Sigma \times Q.$$

С помощью конечного автомата динамическая система строится следующим образом. Множество $\mathcal{A} = \Sigma \times Q$, $\mathrm{T} = \mathbb{N}$. Будем обозначать $\mathcal{A} = \mathcal{A}_\Sigma \times \mathcal{A}_Q$. Отображение p задается по правилу:

$$p(\{\sigma_0, q_0\}, 0) = \{\sigma_0, q_0\},$$

$$p(\{\sigma_0, q_0\}, 1) = \{\sigma_1, q_1\},$$

$$p(\{\sigma_1, q_1\}, 2) = \{\sigma_2, q_2\}$$

и так далее. Важной характерной чертой конечного автомата является то, что задаваемая им динамическая система является периодической, то есть начиная с некоторого номера состояния будут повторятся.

Приступим к реализации класса на C# для описания абстрактной динамической системы.

```
class TT
{
    public TT() { }
}

class TR : TT
{
    public double t;
    public TR()
    {
        t = 0;
    }
```

```
}

class TN : TT
{
    public int n;
    public TN()
    {
        n = 0;
    }
}

abstract class TDS
{
    public object A;
    public TT T;

    public TDS(object A)
    {
        this.A = A;
        T = new TT();
    }

    abstract public void Next();
}

abstract class TRDS : TDS
{
    public TRDS(object A)
        : base(A)
    {
        T = new TR();
```

```
    }

    public double Get_t()
    {
        return ((TR)T).t;
    }

    protected void Set_t(double t)
    {
        ((TR)T).t = t;
    }

    public void Inc(double dt)
    {
        Set_t(Get_t() + dt);
    }
}

abstract class TNDS : TDS
{
    public TNDS(object A)
        : base(A)
    {
        T = new TN();
    }

    public int Get_n()
    {
        return ((TN)T).n;
    }
```

```
        protected void Set_n(int n)
        {
            ((TN)T).n = n;
        }

        public void Inc()
        {
            Set_n(Get_n() + 1);
        }
    }
```

Мы ввели фиктивный класс TT смысл, которого только в том, чтобы быть родителем для классов TR и TN — соответственно для непрерывной и дискретной динамических систем. Мы также сразу создали два наследника класса TDS для различного типа динамических систем.

В качестве первого примера динамической системы мы рассмотрим отображение Хенона. Фазовым множеством здесь является множество $\mathbb{R}^2$, время дискретное, а отображение задается формулой

$$p(x, n) = T^n x,$$

где отображение T определено по формуле

$$T \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 + x_2 - ax_1^2 \\ bx_1 \end{pmatrix},$$

где $a > 0$, $0 < b < 1$. В этой динамической системе возникает, так называемый, странный аттрактор. Движение этой динамической системы носит хаотичный характер. Реализуем и посмотрим на численные результаты.

```
    class THenon : TNDS
    {
```

```
double a, b;
public THenon(double a, double b)
    : base(new double[2])
{
    this.a = a;
    this.b = b;

}

public void Set_x1(double x1)
{
    ((double[])A)[0] = x1;
}

public void Set_x2(double x2)
{
    ((double[])A)[1] = x2;
}

public double Get_x1()
{
    return ((double[])A)[0];
}

public double Get_x2()
{
    return ((double[])A)[1];
}

public override void Next()
{
```

```
            double x1, x2;
            x1 = 1.0 + Get_x2() - a * Get_x1() * Get_x1();
            x2 = b * Get_x1();
            Set_x1(x1);
            Set_x2(x2);

            Inc();
        }

    }
```

Запустим наш класс для расчета первых 10^3 точек, и на рисунке 19.1 приведем фазовый портрет.

```
THenon Henon = new THenon(1.4, 0.3);

StreamWriter F = File.CreateText("henon.txt");
while (Henon.Get_n() <= 10000)
{
    Console.WriteLine("{0}\t{1}", Henon.Get_x1(), Henon.Get_x2());
    F.WriteLine("{0}\t{1}", Henon.Get_x1(), Henon.Get_x2());

    Henon.Next();
}
F.Close();
```

В качестве примера непрерывной динамической системы мы рассмотрим нелинейный осциллятор Ван-дер-Поля. Фазовое пространство этой динамической системы также двумерная плоскость: $\mathcal{A} = \mathbb{R}^2$, время непрерывное, а полугруппа преобразования задается решением задачи Коши для следующего нелинейного дифференциального уравнения второго порядка.

$$y''(t) - a(1 - by^2(t))y'(t) + y(t) = 0.$$

Здесь $a, b \in \mathrm{R}$ — параметры системы. Параметр $a > 0$ называется параметром возбуждения.

Реализуем соответствующий класс на C# .

```
class TPole : TRDS
{
    TPoleRK RK;

    double h;
    public TPole(double a, double b, double h)
        : base(new double[2])
    {
        this.h = h;
        RK = new TPoleRK(a, b);
        Set_x1(10.0);
        Set_x2(10.0);
        RK.SetInit(0, new double[] { Get_x1(), Get_x2() });
    }

    public void Set_x1(double x1)
    {
        ((double[])A)[0] = x1;
    }

    public void Set_x2(double x2)
    {
        ((double[])A)[1] = x2;
    }

    public double Get_x1()
    {
```

```
        return ((double[])A)[0];
    }

    public double Get_x2()
    {
        return ((double[])A)[1];
    }

    public override void Next()
    {
        RK.NextStep(h);
        Inc(h);
        Set_x1(RK.Y[0]);
        Set_x2(RK.Y[1]);
    }
}

class TPoleRK : TRungeKutta
{
    double a, b;
    public TPoleRK(double a, double b) : base(2)
    {
        this.a = a;
        this.b = b;
    }

    public override void F(double t, double[] Y,
    ref double[] FY)
    {
        FY[0] = Y[1];
        FY[1] = a * (1 - b * Y[0] * Y[0]) * Y[1] - Y[0];
```

```
        }
    }
```

Теперь проведем вычислительный эксперимент.

```
TPole Pole = new TPole(1, 0.05, 0.01);

StreamWriter F = File.CreateText("pole.txt");
while (Pole.Get_t() <= 100)
{
    Console.WriteLine("{0}\t{1}\t{2}",
    Pole.Get_t(), Pole.Get_x1(), Pole.Get_x2());
    F.WriteLine("{0}\t{1}\t{2}",
    Pole.Get_t(), Pole.Get_x1(), Pole.Get_x2());

    Pole.Next();
}
F.Close();
```

В результате мы получим фазовый портрет, который представлен на рисунке 19.2. Мы видим, что наша динамическая система имеет так называемый предельный цикл.

Ключевые термины

Абстрактно заданная динамическая система — динамическая система на множестве, заданная с помощью полугруппы преобразований этого множества.

Конечный автомат — абстрактный автомат, моделирующий конечную динамическую систему.

Динамическая система — математическая модель системы, процессы в которой развиваются во времени.

Фазовые координаты — элементы фазового пространства, характеризующие состояние системы.

Краткие итоги: Дано общее определение динамической системы. Реализованы классы для моделирования динамических систем и проведены вычислительные эксперименты. Промоделировано отображение Хенона и нелинейный осциллятор Ван-дер-Поля.

Глава IX

Моделирование игровых ситуаций

20. Объектно-ориентированная реализация агентного моделирования

В этой лекции мы будем рассматривать своеобразные динамические системы. Но в отличии от предыдущей лекции мы не будем задавать отображение фазового пространства. Вместо этого мы будем рассматривать множество **агентов**, которые будут функционировать в заданной **внешней** для них среде.

Перейдем к формальному описанию. Пусть В нашей системе задано N_A агентов, которых мы будем обозначать:

$$A_i,\ i = 1, \ldots, N_A.$$

Мы будем рассматривать лишь конечное количество агентов. Каждый агент имеет свое множество состояний: Q_i — множество состояний агента A_i, $i = 1, \ldots, N_A$. Кроме того будем рассматривать множество состояний внешней среды. Это множество будем обозначать: Q_0. В этом множестве состояний выделено два элемента $q_0, q_f \in Q_0$. Элемент q_0 — это начальное состояние внешней среды, а элемент

q_f — финальное состояние системы. Мы будем рассматривать динамическую систему с дискретным временем $n = 0, 1, 2, \ldots$. Введем обозначения $S(i, n)$:

$$S(i, n) \in Q_i,\ i = 0, 1, \ldots, N_A;\ n = 0, 1, 2, \ldots,$$

смысл функции $S(i, n)$ — это состояние агента (для $i > 0$) или внешней среды (для $i = 0$) в момент n.

Внешняя среда в заданном порядке предоставляет право агентам изменять свое внутреннее состояние и состояние внешней среды. При этом, разумеется агент имеет право менять состояние внешней среды не произвольно, а лишь выбирая это состояние из заданных возможностей. Эти возможности завися как от состояния самого агента и текущего состояния внешней среды. Введем множества возможностей i-го агента, находящегося в состоянии $A_q \in Q_i$, когда внешняя среда находится в состоянии $q \in Q_0$:

$$W_i(A_q, q) \subset Q_0,\ q \in S_0,\ A_q \in S_i,\ i = 1, 2, \ldots, N_A.$$

У каждого агента есть «функция» перехода. Но эта не есть детерминированная функция, а реакция агента. Мы будем это отображение обозначать

$$R_i(S(i, n), S(0, n)) \in W_i(S(i, n), S(0, n)),$$

$$i = 1, 2, \ldots, N_A,\ n = 0, 1, 2, \ldots.$$

При реализации отображения R_i могут быть использованы генераторы случайных чисел, поэтому выбор «хода» агента может носить и стохастический характер.

Необходимо еще выбрать **функцию определения очередности ходов агентов**. Обозначим эту функцию через

$$I : \mathbb{N} \times Q_0 \to \{1, 2, \ldots, N_A\}.$$

Мы подчеркиваем, что выбор игрока, которому предлагается изменить свое внутреннее состояние и состояние внешней среды, зависит от текущего времени (n) и от текущего состояния внешней среды. Часто при разыгрывании ситуации право хода чередуется между агентами. Однако возможны и более сложные случаи.

Мы реализуем игру в «крестики-нолики». Рассматривается квадратное игровое поле 3×3. В игру играют два игрока: первый ставит отметку «крестиком», второй «ноликом». Цель — выстроить три крестика или нолика в один ряд или по диагонали. Игра завершается либо выигрышем одного из игроков, либо вничью, когда нет более свободных клеток.

Поскольку в этой игре известны оптимальные стратегии, при соблюдении которых игра всегда заканчивается вничью, то мы реализуем двух агентов, каждый из которых будем придерживаться разных стратегий.

Стратегия первого игрока. Если свободен центр, ставим играем в центр. Если центр занят, но есть хотя бы один свободный угол, то играем в один из свободных углов, выбирая его случайным образом. Если центр и углы заняты, играем случайным образом в любое свободное поле. Стратегия второго игрока. Всегда играем в любое свободное поле. Интуитивно ясно, что стратегия первого игрока хотя и не является оптимальной, но должна приводить к победе чаще, чем второго.

Мы реализуем данную игру с помощью подхода, основанного на агентного моделирования. Начнем программирование с создания класса, ответственного за игровое поле.

```
class TArray
{
    int[] arr;
    int Count = 9;
```

```
public TArray()
{
    arr = new int[9];
    for (int i = 0; i < 9; i++)
    {
        arr[i] = 0;
    }
}

public bool Check(int i, int Star)
{
    if (Who(i) == 0)
    {
        arr[i] = Star;
        Count--;
        return true;
    }
    else
    {
        return false;
    }

}

public int GetCount()
{
    return Count;
}

public int Who(int i)
```

```
{
    return arr[i];
}

public bool IsOver(int Star)
{
    int i;
    for(i=0;i<3;i++)
    {
        if ((arr[i] == Star) &&
        (arr[i + 1] == Star) && (arr[i + 2] == Star))
        {
            return true;
        }

        if ((arr[i] == Star) &&
        (arr[i + 3] == Star) && (arr[i + 6] == Star))
        {
            return true;
        }
    }

    if ((arr[0] == Star) &&
    (arr[4] == Star) && (arr[8] == Star))
    {
        return true;
    }

    if ((arr[2] == Star) &&
    (arr[4] == Star) && (arr[6] == Star))
    {
```

```
            return true;
        }

        return false;
    }
}
```

Этот класс хранит текущую позицию и умеет найти выигрышную позицию, если она сложилась на игровом поле. Этот класс предоставляет информацию о текущей позиции и возвращает количество свободных клеток.

Следующие классы описывают поведение игроков.

```
abstract class TAgent
{
    public int Star;
    public TAgent(int Star)
    {
        this.Star = Star;
    }

    abstract public int move(TArray Arr, Random rnd);
}

class TAgent1 : TAgent
{
    public TAgent1(int Star) : base(Star) { }
    public override int move(TArray Arr, Random rnd)
    {
        if (Arr.Who(4) == 0)
        {
            Arr.Check(4, Star);
```

```
    return 4;
}

int[] Angles = new int[4] {0, 2, 6, 8};

int AngCount = 0;
int i, I;

for (i = 0; i < 4; i++)
{
    if (Arr.Who(Angles[i]) == 0)
    {
        AngCount++;
    }
}

if (AngCount > 0)
{
    I = rnd.Next(AngCount);

    for (i = 0; i < 4; i++)
    {
        if (Arr.Who(Angles[i]) == 0)
        {
            if (I == 0)
            {
                Arr.Check(Angles[i], Star);
                return Angles[i];
            }
            I--;
        }
```

```
            }
        }

        I = rnd.Next(Arr.GetCount());

        for (i = 0; i < 9; i++)
        {
            if (Arr.Who(i) == 0)
            {
                if (I == 0)
                {
                    Arr.Check(i, Star);
                    return i;
                }
                I--;
            }
        }

        return 0;
    }
}

class TAgent2 : TAgent
{
    public TAgent2(int Star) : base(Star) { }

    public override int move(TArray Arr, Random rnd)
    {
        int I = rnd.Next(Arr.GetCount());

        for (int i = 0; i < 9; i++)
```

```
            {
                if (Arr.Who(i) == 0)
                {
                    if (I == 0)
                    {
                        Arr.Check(i, Star);
                        return i;
                    }
                    I--;
                }
            }

            return 0;
        }
    }
```

И последний класс управляет всей игрой. В начале он выбирает очередность ходов, но первый игрок всегда играет «крестиком» (цифрой 1), а второй «ноликом» (цифрой 2).

```
    class TXO
    {
        public Random rnd;
        TArray Arr;
        TAgent1 A1;
        TAgent2 A2;

        public TXO()
        {
            rnd = new Random();
            Clear();
        }
```

```
void Clear()
{
    Arr = new TArray();
    A1 = new TAgent1(1);
    A2 = new TAgent2(2);
}

public int Run()
{
    Clear();

    int res = 0;

    TAgent[] Agents = new TAgent[2];

    if (rnd.Next(2) == 0)
    {
        Agents[0] = A1;
        Agents[1] = A2;
    }
    else
    {
        Agents[0] = A2;
        Agents[1] = A1;
    }

    for (int i = 0; i < 9; i++)
    {
        if ((i % 2) == 0)
        {
```

```
                Console.WriteLine("{0} => {1}",
                Agents[0].Star,
                Agents[0].move(Arr, rnd));
                if (Arr.IsOver(Agents[0].Star))
                {
                    res = Agents[0].Star;
                    break;
                }
            }
            else
            {
                Console.WriteLine("{0} => {1}",
                Agents[1].Star,
                Agents[1].move(Arr, rnd));
                if (Arr.IsOver(Agents[1].Star))
                {
                    res = Agents[1].Star;
                    break;
                }
            }

        }

        return res;
    }
}
```

Теперь сыграем 1000 партий.

```
TXO XO = new TXO();

int Res1 = 0;
```

```
    int Res2 = 0;

    int Star;

    for (int i = 0; i < 1000; i++)
    {
        Console.WriteLine("\nNew Game");
        Star = XO.Run();
        Console.WriteLine("Winner = {0}", Star);
        if (Star == 1)
        {
            Res1++;
        }
        else
        {
            Res2++;
        }
    }

    Console.WriteLine("1 => {0}\t2 => {1}", Res1, Res2);
}
```

Типичный результат игры такой:

```
..........

 New Game
 1 => 4
 2 => 1
 1 => 6
 2 => 3
 1 => 2
```

```
Winner = 1
1 => 745   2 => 255
```

Примерно в трех из четырех партий первый игрок, обладающий более интеллектуальной стратегией выигрывает у второго, который по сути играет автоматически случайно ставя «нолик».

Ключевые термины

Агент — некоторая сущность способная взаимодействовать с внешней средой.

Внешняя среда — программная среда, в которой действуют агенты.

Функция определения очередности ходов агентов — правило, задающее последовательность ходов агентов.

Краткие итоги Реализована нетривиальная динамическая система на основе агентного моделирования. Проведены вычислительные эксперименты, которые дали статистические характеристики агнетов.

Ключевые термины

Агент — некоторая сущность способная взаимодействовать с внешней средой.

Внешняя среда — программная среда, в которой действуют агенты.

Функция определения очередности ходов агентов — правило, задающее последовательность ходов агентов.

Краткие итоги: Реализована нетривиальная динамическая система на основе агентного моделирования. Проведены вычислительные эксперименты, которые дали статистические характеристики агнетов.

21. Объектно-ориентированный подход к теории игр

На прошлой лекции мы рассматривали игру в «крестики-нолики», в которой принимали участие два различных агента. В настоящей лекции мы рассмотрим элементы теории игр. **Теория игр** — это прикладная математическая дисциплина, в которой изучаются методы нахождения оптимальных решений в условиях неопределенности и ситуациях противодействия со стороны других игроков. Современную математическую форму теория игр приобрела после известного труда Дж. фон Неймана и О. Моргенштерна «Теория игр и экономическое поведение», вышедшего в 1944 году. В настоящее время теория игр находит свое применение в экономических науках, в социальных науках, биологии, а также и в математических дисциплинах, таких как, математическая статистика, функциональный анализ и других.

В отличии от задач оптимизации, где необходимо найти оптимальное решение в заданных условиях, в теории игр необходимо найти оптимальное решение в условиях противодействия со стороны других игроков.

В теории игр рассматривается большое количество постановок различных игр. Мы рассмотрим подробно самую известную постановку в теории игр — **антагонистичную игру двух лиц**.

Обозначим через I множество всех игроков. Мы будем рассматривать конечное число игроков. Мы будем различать игроков по номерам

$$I = \{1, 2, \ldots, N\}.$$

Предположим, что каждый игрок $i \in I$ имеет в своем распоряжении определенное множество стратегий, которое мы обозначим через S_i.

Процедура игры происходит следующим образом: каждый игрок выбирает одну стратегию из своего множества стратегий $s_i \in S_i$.

Вектор выбранных стратегий всех игроков обозначим через

$$s = (s_1, s_2, \ldots, s_N).$$

Вектор s называется **ситуацией в игре**. Множество всех возможных ситуаций можно ввести по формуле

$$S = \prod_{i \in I} S_i.$$

В каждой сложившейся ситуации игроки получают определенные выигрыши. Договоримся считать, что выигрыш может быть и отрицательным, что означает проигрыш. Выигрыш игрока i в ситуации s обозначим через $H_i(s)$. Функция H_i, определенная на множестве всех ситуаций

$$H_i : S \to \mathbb{R}$$

называется функцией выигрыша i-го игрока. Мы будем измерять выигрыши действительными числами, хотя не всегда выигрыш может быть измерен числом.

Бескоалиционной игрой называется система

$$\Gamma = \langle I, \{S\}_{i \in I}, \{H_i\}_{i \in I} \rangle,$$

где I, S_i являются множествами, а H_i — функции на множестве S, принимающие вещественные значения.

Наиболее часто встречается ситуация, когда сумма выигрышей всех игроков во всех ситуациях является постоянной, что соответствует тому, что игроки по сути делят между собой фиксированную сумму. Игра называется игрой с постоянной суммой, если

$$\sum_{i \in I} H_i(s) = const$$

при всех ситуациях $s \in S$.

Мы будем рассматривать антагонистичные игры. Игра называется антагонистичной, если число игроков равно двум, т.е. $I = \{1, 2\}$, а значения функций выигрыша в сумме равны нулю

$$H_1(s) = -H_2(s),\ s \in S.$$

Если в теории оптимизации основной задачей является нахождения оптимальных решений, то в теории игр аналогом этого является нахождения ситуации равновесия. Ситуация $s^* \in S$ называется **ситуацией равновесия в игре**, если ни одному из игроков не выгодно отступать от этой стратегии. формально это можно записать следующей формулой $s^* = (s_1^*, s_2^*)$

$$H_1(s_1, s_2^*) \leq H_1(s_1^*, s_2^*) \leq H_1(s_1^*, s_2),\ s \in S.$$

Если множества стратегий конечны, то антагонистичные игры удобно записывать в матричном виде. Пусть множество стратегий первого игрока равно $n > 1$, а второго — $m > 1$, тогда запишем в виде матрицы значения функции выигрышей

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

Игра в этом случае состоит в том, что первый игрок выбирает строку, а второй игрок (одновременно!) выбирает столбец. Число, стоящее на пересечении выбранных строки и столбца, означает выигрыш первого игрока и проигрыш второго игрока.

В матричной игре ситуация (i^*, j^*) называется равновесной, если

$$a_{ij^*} \leq a_{i^*j^*} \leq a_{i^*j}$$

для всех $i = 1, \dots, m$ и $j = 1, \dots, n$. В теории игр доказывается, что для существования ситуации равновесия необходимо и достаточно,

чтобы было выполнено равенство

$$\max_i \min_j a_{ij} = \min_j \max_i a_{ij} = c.$$

Число c в этом случае называется ценой игры. Если бы в каждой игре существовала бы ситуация равновесия, то игры бы не имели смысл. К счастью или к сожалению, но во многих играх ситуации равновесия не существует. Самый простой пример — игра в «чет–нечет». Матрица этой игры такова

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Фундаментальным результатом теории игр является тот факт, что любая матричная игра имеет ситуацию равновесия в смешанных стратегиях. **Смешанной стратегией** называется случайная величина, значениями которой являются стратегии игрока. Смешанная стратегия — это распределение вероятностей на множестве допустимых стратегий, которую можно представить вектором с неотрицательными компонентами, сумма которых равна единице.

При смешанном расширении понятия матричной игры, игроки выбирают свои смешанные стратегии: первый игрок

$$X = (x_1, \ldots, x_m),\ x_i \geq 0,\ \sum_{i=1}^{m} x_i = 1,$$

$$Y = (y_1, \ldots, y_m),\ y_i \geq 0,\ \sum_{i=1}^{m} y_i = 1,$$

Выигрыш в смешанных расширениях рассчитывается как математическое ожидание. Выигрыш первого игрока равен

$$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_i y_j$$

Теорема 21.1. *В матричной игре с матрицей выигрышей A имеет место*

$$\max_X \min_j (X A_{\cdot j}) = \min_Y \max_i (A_{i\cdot} Y.)$$

При чем внешние экстремумы достигаются на оптимальных смешанных стратегиях.

В этой теореме $A_{i\cdot}$ обозначает i-ую строку, а $A_{j\cdot}$ — j-ый столбец.

Несмотря на наличие этой теоремы вопрос конкретного определения оптимальных стратегий является очень сложным.

Мы в нашем курсе проведем моделирование матричной игры и проверим ряд известных решений некоторых игр. Начнем с программирования класса матричной игры.

```
class TGame
{
    protected double[,] A;
    protected int m = 0, n = 0;
    Random rnd;
    public TGame()
    {
        rnd = new Random();
    }

    public double Calc(double[] X, double[] Y, int Count)
    {
        double res = 0;
        int i, j;
        for (int k = 0; k < Count; k++)
        {
            i = Release(X, m);
            j = Release(Y, n);
```

```
            res += GetAij(i, j);
        }

        return res / (double)Count;
    }

    public int Release(double[] Z, int N)
    {
        double p = rnd.NextDouble();
        double a = 0;

        for (int i = 1; i <= N; i++)
        {
            a += Z[i];
            if (p <= a)
            {
                return i;
            }
        }
        return N;
    }

    public double GetAij(int i, int j)
    {
        return A[i, j];
    }

    public double GetC(double[] X, double[] Y)
    {
        double res = 0;
```

```
            int i, j;

            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    res += A[i, j] * X[i] * Y[j];
                }
            }

            return res;
        }
    }
```

Теперь создадим два наследных класса, в которых мы реализуем две матричные игры.

```
    class TGame1 : TGame
    {
        public TGame1()
            : base()
        {
            m = 2;
            n = 2;
            A = new double[3, 3];
            A[1, 1] = 0;
            A[1, 2] = 1;
            A[2, 1] = 1;
            A[2, 2] = 0;
        }
    }
```

```
class TGame2 : TGame
{
    public TGame2()
        : base()
    {
        m = 3;
        n = 3;
        A = new double[4, 4];
        A[1, 1] = 0;
        A[1, 2] = 1;
        A[1, 3] = -2;
        A[2, 1] = -1;
        A[2, 2] = 0;
        A[2, 3] = 3;
        A[3, 1] = 2;
        A[3, 2] = -3;
        A[3, 3] = 0;
    }
}
```

Класс $TGame1$ — это самая простая нетривиальная игра. По сути это игра в «чет–нечет». В этой игре нет равновесных чистых стратегий, а в смешанных стратегиях эта игра имеет следующее решение

$$X = \left(\frac{1}{2}, \frac{1}{2}\right),$$

$$Y = \left(\frac{1}{2}, \frac{1}{2}\right).$$

Вторая игра, реализованная в классе $TGame2$, представляет собой

более сложную игру со следующей платежной матрицей

$$A = \begin{pmatrix} 0 & 1 & -2 \\ -1 & 0 & 3 \\ 2 & -3 & 0 \end{pmatrix}$$

В этой игре также нет состояния равновесия, но есть решение в смешанных стратегиях одинаковое для обоих игроков:

$$X = \left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right),$$

$$Y = \left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right).$$

Цена этой игры равна нулю.

Проверим эти решения с помощью наших классов.

```
double[] X;
double[] Y;

TGame1 Game1 = new TGame1();
X = new double[3] {0, 0.5, 0.5 };
Y = new double[3] {0, 0.5, 0.5 };

Console.WriteLine("Game1: theory = {0}, Res = {1}",
Game1.GetC(X, Y), Game1.Calc(X, Y, 1000000));

TGame2 Game2 = new TGame2();
X = new double[4] { 0, 0.5, 1.0 / 3.0, 1.0 / 6.0 };
Y = new double[4] { 0, 0.5, 1.0 / 3.0, 1.0 / 6.0 };

Console.WriteLine("Game2: theory = {0}, Res = {1}",
Game2.GetC(X, Y), Game2.Calc(X, Y, 1000000));
```

После запуска мы получим примерно следующее:

```
Game1: theory = 0.5, Res = 0.500092

Game2: theory = 0, Res = -0.00097
```

Ключевые термины

Антагонистичная игра — игра двух игр с нулевой суммой.

Ситуация в игре — набор выбранных стратегий всех игроков.

Ситуация равновесия — такая ситуация, при которой ни один из игроков не заинтересован в изменении стратегии.

Смешанная стратегия — случайная величина, значениями которой являются стратегии игрока.

Теория игр — прикладная математическая дисциплина, в которой изучаются методы нахождения оптимальных решений в условиях неопределенности и ситуациях противодействия со стороны других игроков.

Краткие итоги: Рассмотрены постановки игр. Для матричных игр приведено объектно-ориентированное моделирование игр. С помощью статистического моделирования исследованы некоторые матричные игры.

22. Вычислительные эксперименты в моделировании волн-убийц

На последней лекции мы рассмотрим реальные вычислительные эксперименты, которые были проведены в Институте океанологии им. П.П. Ширшова РАН совместно с Физическим институтом им. П.Н. Лебедева РАН и Университетом Аризоны (США). Были проведены масштабные вычислительные эксперименты по моделированию волн-убийц в мировом океане.

Под волнами-убийцами в океанологии понимаются внезапные поверхностные волны экстремальной амплитуды. В последние несколь-

ко лет были получены экспериментальные данные, подтверждающие существование этого феномена. Волны-убийцы наносят серьезный ущерб морскому судоходству и морским нефтяным платформам. Особую опасность волнам-убийцам придает внезапность их возникновения и огромная энергия этих волн. Обычно амплитуда волн-убийц достает 20-30 метрового значения.

После получения натурных данных началось систематическое изучение этого эффекта. В лаборатории нелинейных волновых процессов Института океанологии им. П.П. Ширшова РАН проводились систематические вычислительные эксперименты по изучению волн-убийц. С помощью доказательных вычислений было показано, что волны-убийцы могут возникать в ходе нелинейной динамики идеальной жидкости со свободной поверхностью. Опишем последние вычислительные эксперименты.

Мы решали численно уравнения Эйлера, описывающие глубокую идеальную жидкость со свободной поверхностью в двумерной геометрии $0 < x < 2\pi$, $-\infty < y < \eta(x)$. Граничные условия на концах интервала $x = 0, 2\pi$, предполагались периодическими.

Течение предполагалось потенциальным, а жидкость несжимаемой,

$$v = \nabla\phi, \quad \operatorname{div} v = 0.$$

Так что потенциал подчинялся уравнению Лапласа

$$\Delta\phi = 0.$$

Мы осуществляли конформное отображение области, занятой жидкостью на нижнюю полуплоскость, координаты на которой $w = u + iv$. Отображение задается функцией $z = z(w)$, $z = x + iy$.

Динамические уравнения формулируются для переменных Дьяченко

$$R = \frac{1}{z'_w} \quad V = i\frac{\partial\Phi}{\partial z}$$

и имеют вид

$$\begin{gathered} R_t(u,t) = i(UR_u - U_u R), \\ V_t(u,t) = i(UV_u - B_u R) + g(R-1), \\ U = P(VR^* + RV^*), \\ B = P(VV^*), \end{gathered} \tag{IX.1}$$

где P — оператор проектирования на нижнюю полуплоскость $P = \frac{1}{2}(1+iH)$, H — аналог оператора Гильберта для периодического случая

$$H[f](y) = \frac{1}{2\pi} v.p. \int_0^{2\pi} \frac{f(u')}{\mathrm{tg}(\frac{u'-u}{2})} du'$$

Система (IX.1) в настоящее время широко используется.

В наших экспериментах начальные условия определялись как ансамбль бегущих в одну сторону волн со средним значением волнового числа $K_0 = 25$.

Мы предполагали, что начальное возмущение поверхности задается суммой гармоник со случайными фазами

$$\eta_0(x) = \sum_{-\frac{1}{2}K_{max}}^{\frac{1}{2}K_{max}} \phi(k-k_0)\cos(kx - \xi_k) \tag{IX.2}$$

Здесь K_{max} — полное число спектральных мод, ξ_k — случайная величина, равномерно распределенная на интервале $-\frac{1}{2}K_{max} < k < \frac{1}{2}K_{max}$.

Начальные значения поля скоростей предполагались связанными с (IX.2) формулами линейной теории. Конформное преобразование осуществлялось при помощи итерационного алгоритма, предложенного А.И.Дьяченко.

Функция $\phi(k)$ определялась по формуле

$$\phi(k) = \begin{cases} \delta_k, & |k| > K_w; \\ \kappa \exp(-\alpha k^2) + \delta_k, & |k| \le K_w. \end{cases} \tag{IX.3}$$

Здесь δ_k — независимые случайные параметры равномерно распределенные на интервале $-\frac{1}{2}K_{max} < k < \frac{1}{2}K_{max}$.

Число $1 \leq K_w \leq 10$ определяло спектральную ширину, κ, α — «внутренние» параметры спектра, определенные так, чтобы «внешние» параметры — средняя крутизна μ

$$\mu^2 = \frac{1}{2\pi}\int_0^{2\pi} \eta_x^2 dx$$

и дисперсия D

$$D = \left(\int_{-K_w}^{K_w} k^2 e^{-\alpha k^2} dk\right)\left(\int_{-K_w}^{K_w} e^{-\alpha k^2} dk\right)^{-1}$$

принимали заданные значения. Далее, мы вычисляем точные значения полной энергии E и следили за тем, чтобы вклад в нее случайного шума составлял не более трех процентов. Было проделано 5000 «элементарных» экспериментов. В каждом эксперименте время менялось в интервале $0 < t < 200$, что соответствовало приблизительно 500 периодам волн. Если происходило обрушение волн, счет прекращался досрочно. В расчетах полное число гармоник было $K_{max} = 2048$ или $K_{max} = 4096$ в зависимости от полной энергии, которая менялась в пределах $1.5 \cdot 10^{-4} \leq E \leq 4 \cdot 10^{-4}$.

Регистрация волн-убийц производилась следующим образом. После окончания элементарного эксперимента рассчитывалась величина ν

$$\nu = \frac{\max \eta(x,t)}{\langle|\eta|\rangle}$$

Здесь максимум в числителе берется по координате и по времени за интервал $0 < t < T$,

$$\langle|\eta|\rangle = \frac{1}{T}\int_0^T \max_{x\in(0,2\pi)} |\eta(x,t)| dt.$$

«Волна-убийца» фиксировалась если параметр ν превышал критическое значение $\nu = 1.8$. Данное определение количественно лишь не существенно отличается от общепринятого, когда считается, что волны-убийцы вдвое превышают существенную высоту (significant wave height) Требовалось также, чтобы локальная крутизна волны $|\eta_x|$ превышала критическое значение $\max\limits_{0<x<2\pi} |\eta_x| \leq 0.3$. Это требование вызвано очевидными физическими соображениями и является весьма существенным.

Результаты экспериментов приведены в таблице 22.1. По горизонтали отложены значения дисперсии, по вертикальной — значения квадрат крутизны. Число «активных» мод начального условия для каждого эксперимента также показано. Из наших данных следует, что даже для волн довольно умеренной крутизны ($\mu^2 \simeq 2.06 \cdot 10^{-3}$, $\mu \simeq 0.045$) образование экстремальной волны за столь короткий отрезок времени как 500 периодов (при периоде 10 секунд это меньше полутора часов) есть весьма вероятное событие даже, если спектральная ширина по волновым числам сравнима с несущим волновым числом. Собственно, этот эксперимент и подчеркивает «обыденность» экстремальных волн. На рисунке 22.1 приведен профиль начальной волны со средней крутизной — $\mu^2 = 2.56 \cdot 10^{-3}$ и дисперсией $D = 4$. На рисунке 22.2 показан профиль волны-убийцы для этой волны. Время образование волны-убийцы — $t = 67.2$, параметр $\nu = 2.13$ максимальная крутизна — 0.558. На рисунке 22.3 приведена плотность импульса в момент образования этой волны-убийцы.

Интересно, что вероятность возникновения экстремальных волн, рассматриваемая как функция от средней крутизны при заданной дисперсии имеет максимум при весьма умеренных крутизнах ($\mu^2 = 2.0 \cdot 10^{-3}$) а затем убывает при увеличении крутизны. Этот факт объясняется увеличением силы конкурирующего эффекта — обрушения волн. Использованная нами схема счета позволяет вести эксперимент только до первого обрушения. Мы полагаем, что при использовании

Таблица 22.1

D	$\mu^2 = 1.54 \cdot 10^{-3}$	$\mu^2 = 2.06 \cdot 10^{-3}$	$\mu^2 = 2.56 \cdot 10^{-3}$	$\mu^2 = 3.08 \cdot 10^{-3}$
$D = 0.07$ $K_w = 1$	0.141	0.638	0.828	0.849
$D = 2$ $K_w = 1$	0.152	0.457	0.616	0.554
$D = 4$ $K_w = 2$	0.011	0.231	0.346	0.272
$D = 6$ $K_w = 3$	0.000	0.192	0.305	0.246
$D = 8$ $K_w = 4$	0.011	0.154	0.280	0.195
$D = 10$ $K_w = 5$	0.022	0.125	0.247	0.186
$D = 12$ $K_w = 6$	0.010	0.173	0.256	0.172
$D = 14$ $K_w = 7$	0.000	0.058	0.216	0.170
$D = 16$ $K_w = 8$	0.000	0.136	0.208	0.151
$D = 18$ $K_w = 9$	0.000	0.118	0.219	0.134
$D = 20$ $K_w = 10$	0.034	0.127	0.206	0.099

более совершенных методик, зависимость вероятности возникновения экстремальных волн от крутизны остается монотонной.

На рисунке 22.4 приведены частоты возникновения волн-убийц в зависимости от дисперсии: $\mu^2 = 1.54 \cdot 10^{-3}$ — сплошная линия, $\mu^2 = 2.56 \cdot 10^{-3}$ — «тире», $\mu^2 = 2.06 \cdot 10^{-3}$ — точечная линия, $\mu^2 = 3.08 \cdot 10^{-3}$ — «точка-тире».

Ключевые термины

Волна-убийца — поверхностная волна экстремальной амплитуды в океане.

Уравнение Эйлера — основное уравнение, описывающее динамику идеальной жидкости.

Краткие итоги: Дано описание вычислительных экспериментов. Приведены результаты о статистических характеристик волн-убийц.

Заключение

В нашем курсе мы рассмотрели некоторые современные численные методы и продемонстрировали их с помощью объектно-ориентированного программирования на языке C# . Мы попытались показать, что для программирования научных задач использование современных объектно-ориентированных подходов оказывается очень удобным.

Скорость прогресса компьютерных технологий существенно превышает скорость прогресса в теории вычислительных методов. На повестке дня стоит конструирование новых, специально спроектированных для объектно-ориентированного программирования, численных методов. И нет никаких сомнений, что в нашем веке наиболее востребованной областью математики будет именно вычислительная математика. Однако для этого теоретическая вычислительная математика должна отвечать современным вызовам. Мы уверены, что такие технологии программирования, как объектно-ориентированное и функциональное программирование, могут обогатить не только практику научных вычислений, но и теоретические подходы.

Вычислительная математика — это наука молодых, поэтому ваши способности будут востребованы в нашей науке во всем мире!

Общий глоссарий

Абстрактная задача Коши — дифференциальное уравнение для функций со значениями в банаховом пространстве с заданным начальным условием.

Абстрактная функция — однозначное отображение одно множества в другое.

Абстрактный класс — класс, содержащий нереализованные методы, которые необходимо реализовать в классах наследниках.

Абстрактный ряд Фурье — аналог ряда Фурье в гильбертовом пространстве.

Абстрактно заданная динамическая система — динамическая система на множестве, заданная с помощью полугруппы преобразований этого множества.

Агент — некоторая сущность способная взаимодействовать с внешней средой.

Алгоритм — последовательность определенных действий, позволяющая за конечное время придти к заданному результату.

Аналитико-числовые методы — численные методы которые содержат операции, выполняемые с использованием аналитического представления математических объектов.

Антагонистичная игра — игра двух игр с нулевой суммой.

Банахово пространство — полное нормированное пространство.

Внешняя среда — программная среда, в которой действуют агенты.

Волна-убийца — поверхностная волна экстремальной амплитуды в океане.

Вычислительная неустойчивость — неустойчивость при работе вычислительной процедуру в следствии машинной арифметики.

Вычислимая последовательность чисел — числовая последовательность, каждый элемент которой может быть получен с помощью вычислимой функции по номеру.

Гильбертово пространство — полное предгильбертово пространство.

Гильбертово пространство $L_2(0, \pi)$ — наиболее известное гильбертово функциональное пространство, состоящее из измеримых функций, интегрируемых в квадратом модуля.

Глобальное решение — решение задачи Коши, существующее при всех $t \geq 0$.

Диагональный оператор — оператор, который можно задать путем умножения коэффициентов абстрактного ряда Фурье на числа.

Динамическая система — математическая модель системы, процессы в которой развиваются во времени.

Задача Коши — дифференциальное уравнение с заданными начальными условиями на решение.

Интерполяционная функция — функция, которая реализует интерполяцию.

Интерполяция — процедура приближенного вычисления значений функции по заданным значениям функции в узловых точках.

Класс — пользовательский тип данных, который включает в себя как данные, так и программный код в виде функций.

Конструктор — метод, который всегда вызывается при создании экземпляра класса.

Конечный автомат — абстрактный автомат, моделирующий конечную динамическую систему.

Конструктивная операция — операция, которая может быть выполнена механически в конечное время и при использовании конечного алфавита.

Конструктивное действительное число — число, являющееся вычислимым пределом вычислимой последовательности рациональных чисел.

Конструктивная функция — алгоритм сопоставляющий одному конструктивному действительному числу другое конструктивное действительное число.

Корень функции — точка, в которой функция принимает нулевое значение.

Корректность задачи по Адамару — существование единственного решения у задачи и непрерывная зависимость решения от данных задачи.

Краевая задача — дифференциальное уравнение с заданными условиями на решение на концах отрезка.

Кубические сплайны — сплайны, которые являются кубическими многочленами между соседними узловыми точками и являются дважды непрерывно дифференцируемыми.

Линейный оператор — линейное отображение одного линейного пространства в другое.

Машина Поста — гипотетическая вычислительная машина, эквивалентная машине Тьюринга.

Машинное ε — такое положительное число ε, для которого на рассматриваемой машине является верным утверждение $1 = 1 + \varepsilon$.

Метод Галеркина — общий метод для приближенного нахождения решений операторных уравнений.

Метод Гаусса — основной метод нахождения решений системы линейных алгебраических уравнений с помощью последовательного исключения неизвестных.

Метод Ньютона — основной итерационный метод для нахожде-

ния приближенного решения системы нелинейных уравнений.

Метод прогонки — метод для обращения трехдиагональной матрицы.

Метод Рунге-Кутта — наиболее распространенный метод численного интегрирования задачи Коши с точностью четвертого порядка.

Метод секущих — двухшаговый итерационный метод решения нелинейных уравнений без необходимости вычислять производную.

Метод Холецкого — метод для обращения симметричных положительно определенных матриц.

Метод Эйлера — простейший метод численного интегрирования задачи Коши с точностью первого порядка.

Метрика — функция на паре элементов метрического пространства, соответствующий аналог расстояния между элементами.

Метрическое пространство — абстрактное пространство в котором введена метрика.

Множество допустимых управлений — множество, которому принадлежит управление.

Наследование — технология проектирования классов, позволяющая создавать новые классы на базе предыдущих классов.

Научное программирование — стиль программирования, ориентированный на научные расчеты.

Начальное приближение — начальное значение последовательности, которая рассчитывается с помощью рекуррентной процедуры.

Начальное условие — функция, которой равно решение в начальный момент.

Невырожденная матрица — квадратная матрица с определителем отличным от нуля.

Невязка приближенного решения — количественная мера неудовлетворения приближенным решением уравнению.

Непрерывный оператор — непрерывное отображение одного

метрического пространства в другое.

Норма — функция на элементах нормированного пространства, аналог длинны.

Нормированное пространство — линейное пространство, в котором введена норма.

Область определения функции — множество на, котором определена функция.

Облачные вычисления — вычисления, спроектированные таким образом, чтобы абстрагироваться от места проведения расчетов.

Операторное уравнение — уравнение в абстрактных пространства записанное с помощью операторов в этих пространствах.

Парадигма объектно-ориентированного программирования — использование объектов и классов, а также технологий инкапсуляции, наследования и полиморфизма.

Парадигма процедурного программирования — систематическое использование отдельных небольших модулей (подпрограмм или процедур.

Парадигма структурного программирования — метод разработки программ «сверху вниз», использование специальных правил хорошего тона при программировании.

Парадигма языков высокого уровня — использование машинно-независимых языков программирования, запись программ с помощью операторов языка программирования а не машинных команд.

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Предгильбертово пространство — линейное пространство, в котором введено скалярное произведение.

Принцип неподвижной точки отображение — существование такого элемента для оператора, который отображается этим оператором в себя.

Проекционный метод — метод аппроксимации эволюционных

уравнений в частных производных системами обыкновенных дифференциальных уравнений.

Разбиение отрезка — конечное множество точек, принадлежащих заданному отрезку.

Распределенные системы — бесконечно мерные системы дифференциальных уравнений.

Ситуация в игре — набор выбранных стратегий всех игроков.

Ситуация равновесия — такая ситуация, при которой ни один из игроков не заинтересован в изменении стратегии.

Скалярное произведение — функция на паре элементов предгильбертова пространства, удовлетворяющая аксиомам скалярного произведения.

Смешанная стратегия — случайная величина, значениями которой являются стратегии игрока.

Спецификаторы доступа — ключевые слова языка C# для указания уровня доступа к полю или методу класса.

Сплайн — интерполяционная функция, производные которой могут иметь разрывы в узловых точках.

Ссылка — именованный адрес в памяти, где хранится переменная, для доступа к этой переменной.

Тезис Черча-Тьюринга — утверждение, что любая интуитивно вычислимая функция может быть вычислена с помощью машины Тьюринга.

Теория игр — прикладная математическая дисциплина, в которой изучаются методы нахождения оптимальных решений в условиях неопределенности и ситуациях противодействия со стороны других игроков.

Узловые точки — элементы из множества разбиения отрезка.

Управляемая система — система, описывающаяся дифференциальным и уравнениями, содержащие управление.

Уравнение Эйлера — основное уравнение, описывающее дина-

мику идеальной жидкости.

Условие Липшица — условие на приращение функции, более сильное чем условие непрерывности, но слабее чем условие дифференцируемости.

Фазовые координаты — элементы фазового пространства, характеризующие состояние системы.

Функция определения очередности ходов агентов — правило, задающее последовательность ходов агентов.

Целевой функционал — функционал на решении, с помощью которого определяется оптимальное решение.

Число обусловленности матрицы — числовая характеристика вычислительной сложности обращения матрицы.

Экземпляр класса — переменная типа класс.

Экстраполяция — процедура вычисления значений функции вне отрезка, на котором задана функция.

Литература

[1] Амосов А.А., Дубинский Ю.А., Н.В. Копченова. Вычислительные методы. — М.: Издательсктй дом МЭИ, 2008.

[2] Бабенко К.И. Основы численного анализа. — М.: Наука, 1986.

[3] Волков Е.А. Численные методы. — СПб.: Издательство «Лань», 2008.

[4] Воробьев Н.Н. Теория игр. Лекции для экономистов-кибернетиков. — Л.: Изл-во Ленингр. ун-та, 1974.

[5] Меньшиков И.С. Лекции по теории игр и экономическому моделированию. — М.: МЗ Пресс, 2006.

[6] Нэш Т. C# 2008: ускоренный курс для профессионалов. — М.: ООО "И.Д. Вильямс 2008.

[7] Степаньянц Г.А. Теория динамических систем. — М.: Книжный дом «Либроком», 2010.

[8] Троелсен Э. C# и платформа .NET. Библиотека программиста. — СПб.: Питер, 2007.

[9] Тыртышников Е.Е. Методы численного анализа. — М.: Издательский дом "Академия 2007.

[10] Форсайт Дж., Малькольм М., Моулер К. Машинные методы математических вычислений. — Издательство «Мир», 1980.

[11] Шамин Р.В. Вычислительные эксперименты в моделировании поверхностных волн в океане. — М.: Наука, 2008 - 136 с.

[12] *Шамин Р.В.* Функциональный анализ от нуля до единицы. М.: ЛЕНАНД/URSS, 2016.

[13] *Шамин Р.В.* Математические вопросы волн-убийц. М.: ЛЕНАНД/URSS, 2016.

[14] *Шамин Р.В.* Полугруппы операторов. М.: РУДН, 2008.

[15] Эльсгольц Л.Э. Качественные методы в математическом анализе. — М.: КомКнига, 2010.

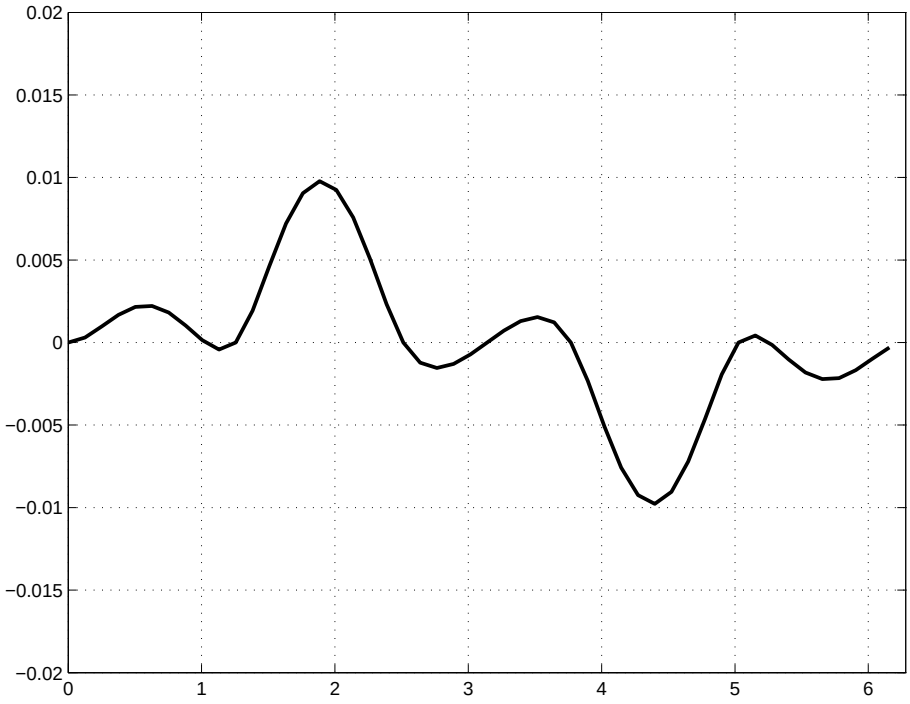
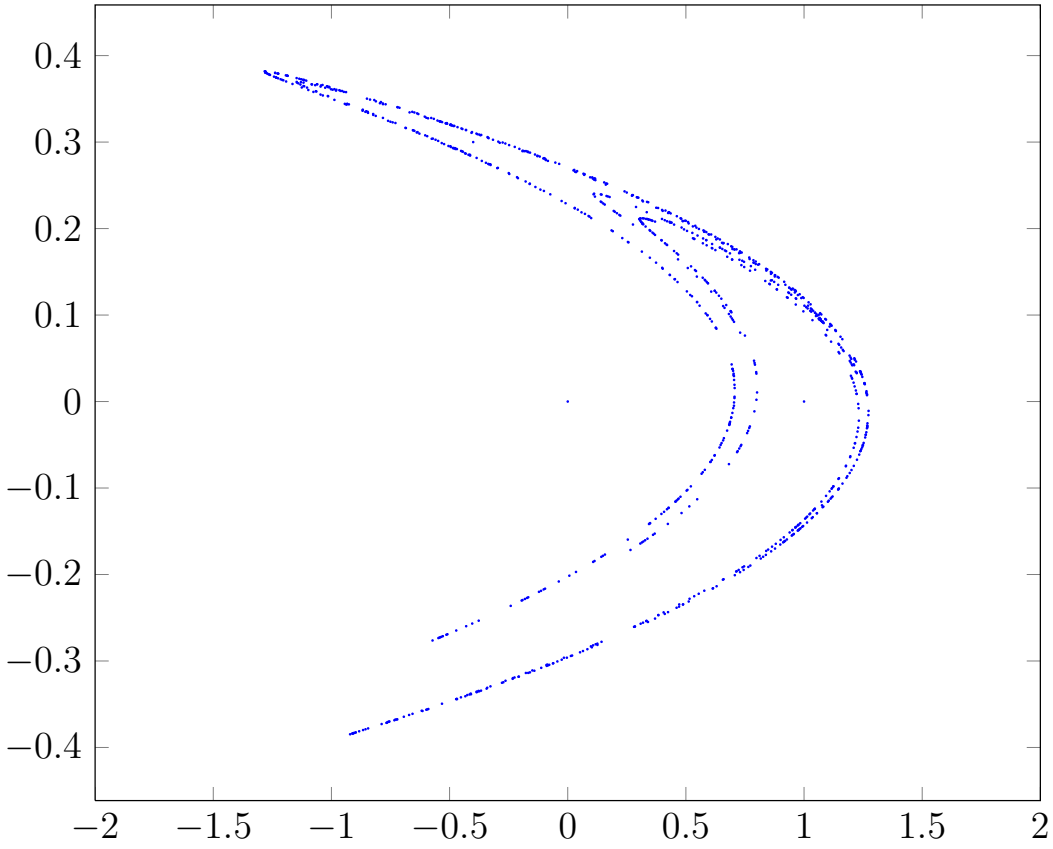


Рис. IX.1. Рис. 15.1 Погрешность кубического сплайна

Рис. 19.1 Множество Хенона



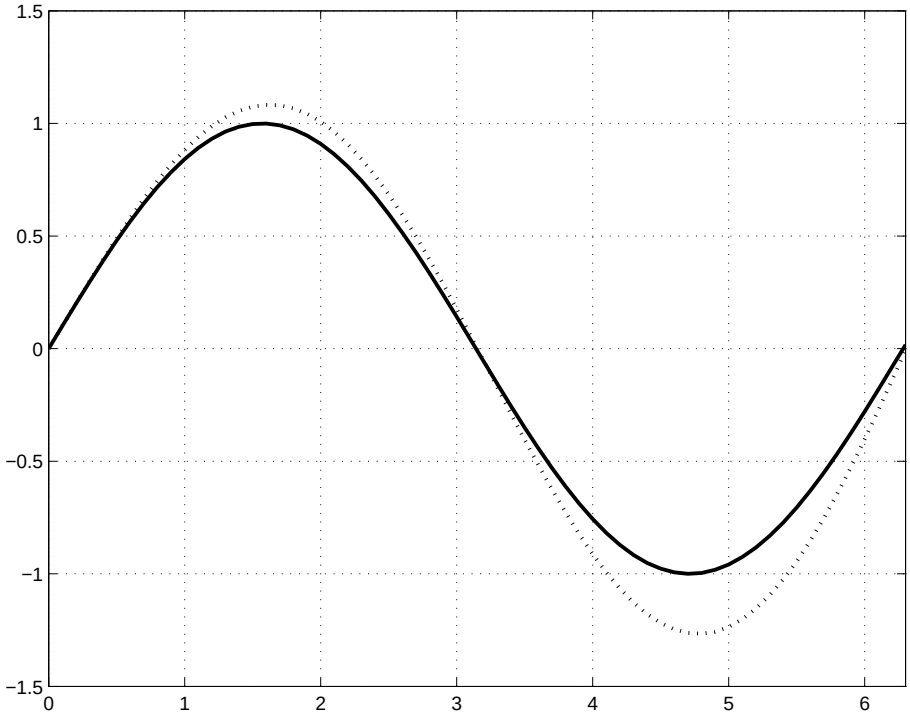


Рис. IX.2. Рис. 16.1 Точное и приближенное решение дифференциального уравнения

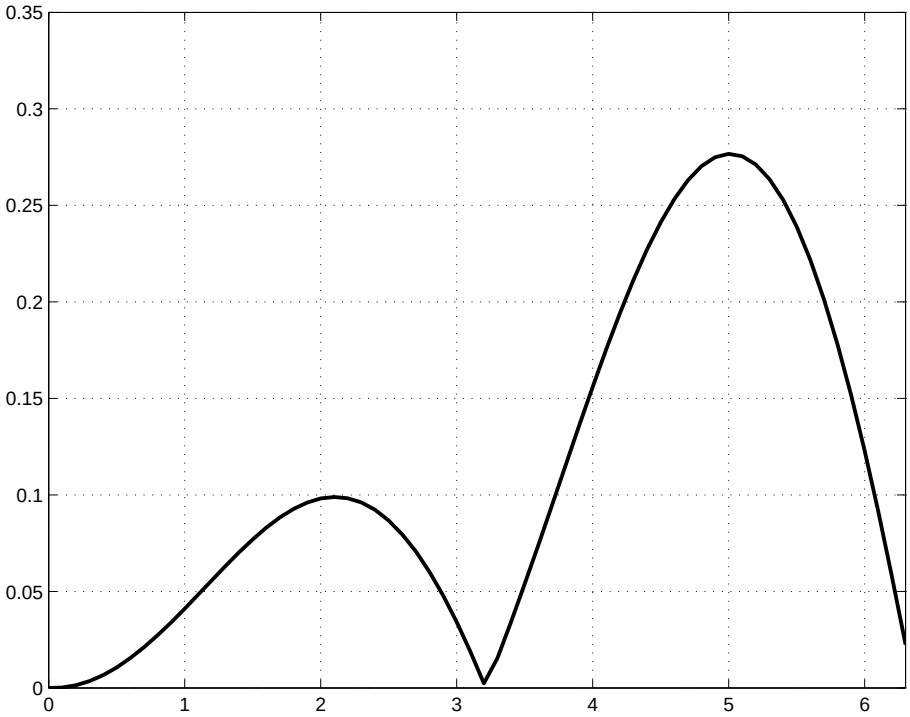


Рис. IX.3. Рис. 16.2 Погрешность метода Эйлера

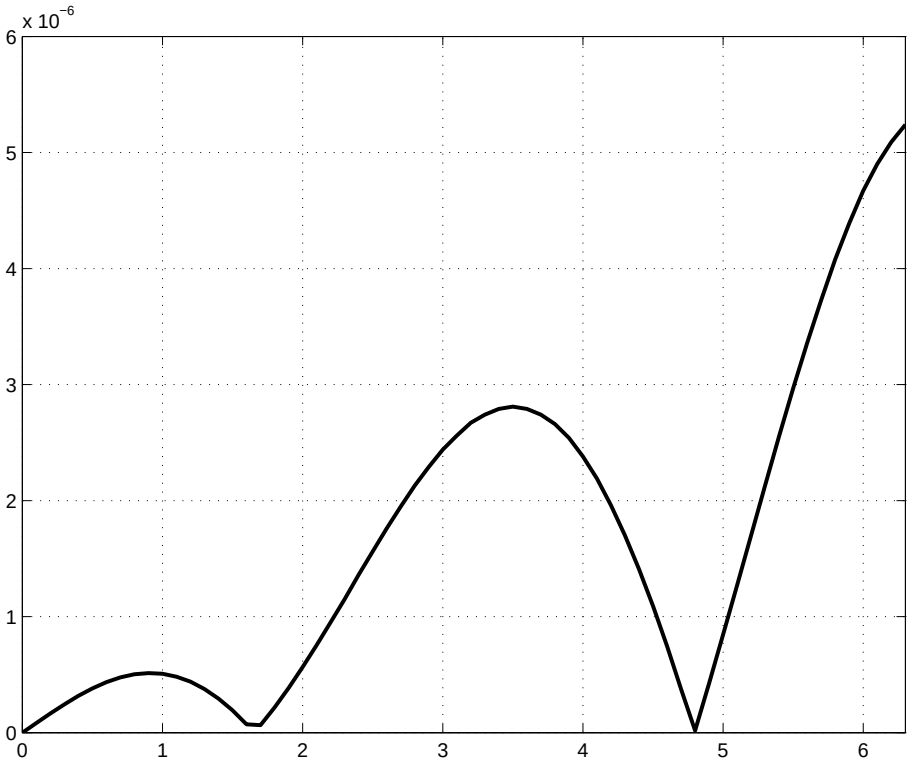


Рис. IX.4. Рис. 16.3 Погрешность метода Рунге–Кутты

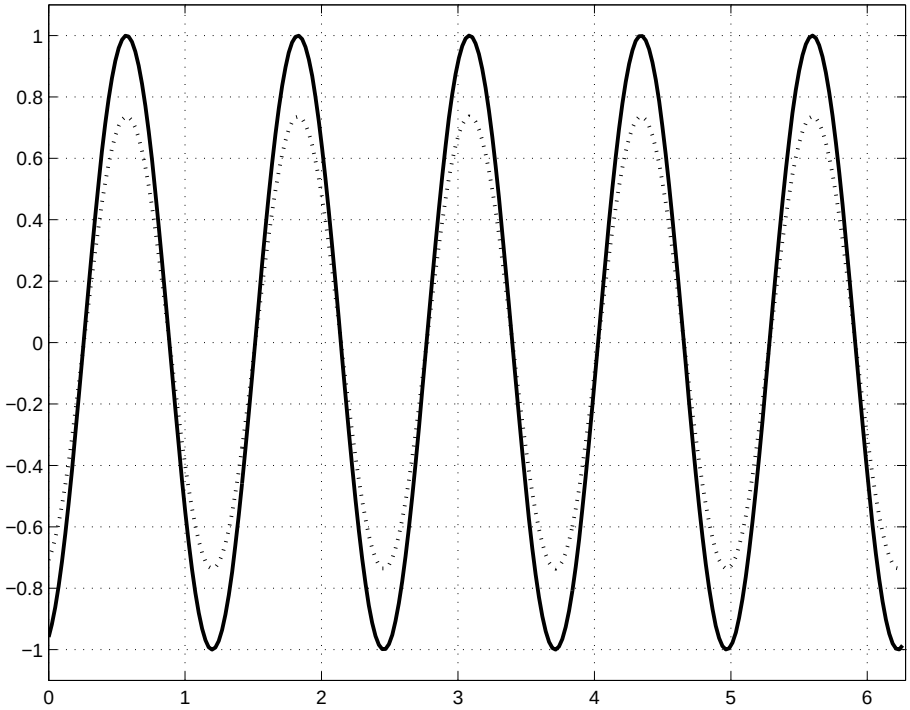


Рис. IX.5. Рис. 17.1 Решение уравнения линейного переноса с помощью кусочно-линейной интерполяции

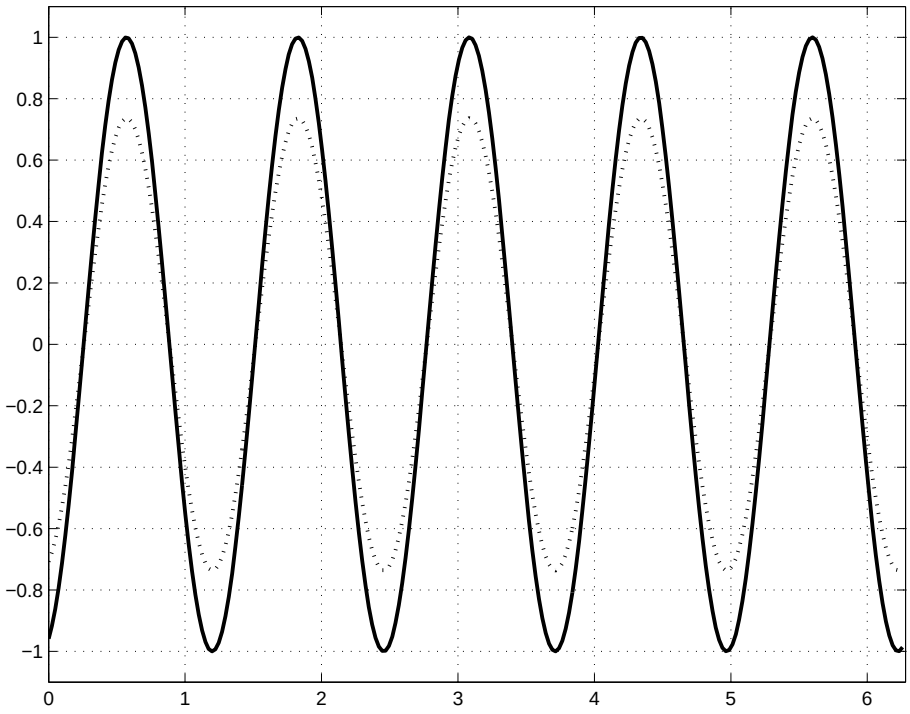


Рис. IX.6. Рис. 17.2 Погрешность приближенного решения уравнения линейного переноса с помощью кусочно-линейной интерполяции

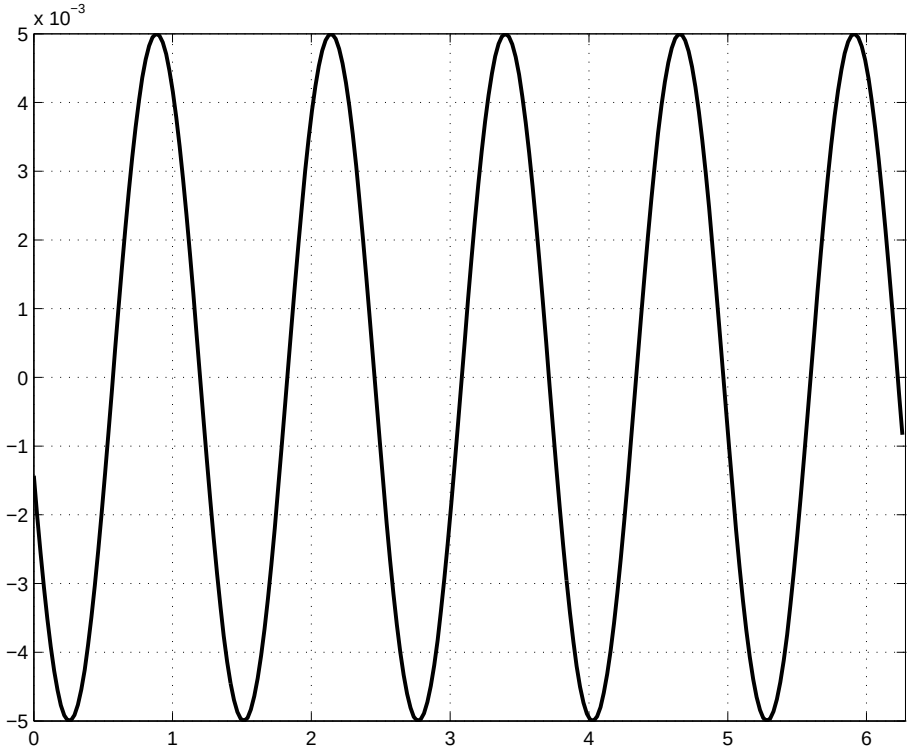


Рис. IX.7. Рис. 17.3 Погрешность приближенного решения уравнения линейного переноса с помощью рядов Фурье

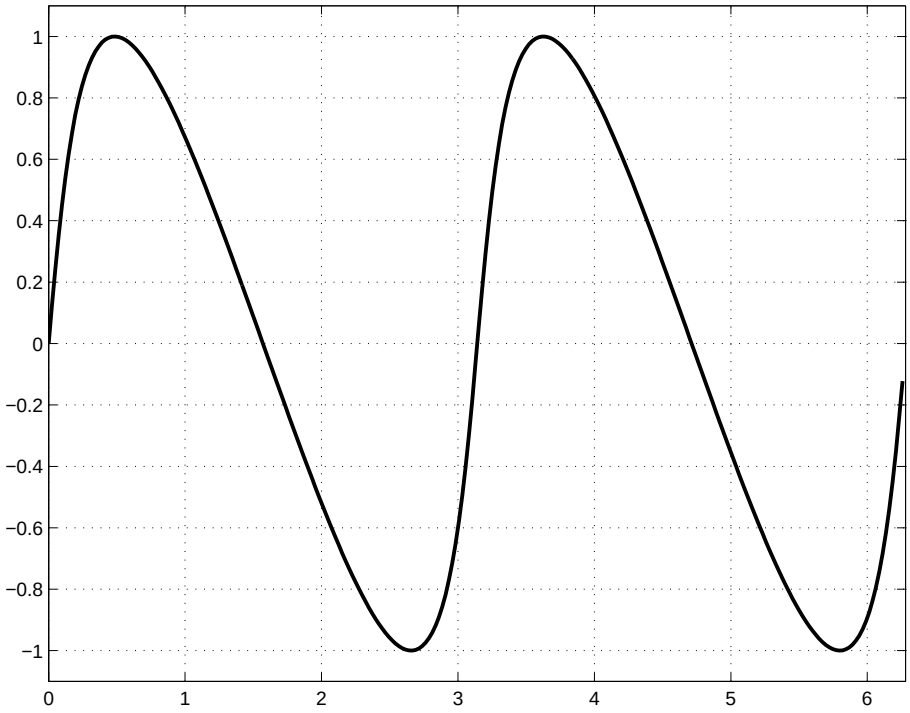


Рис. IX.8. Рис. 17.4 Приближенное решение уравнения нелинейного переноса при $t = 0.3$

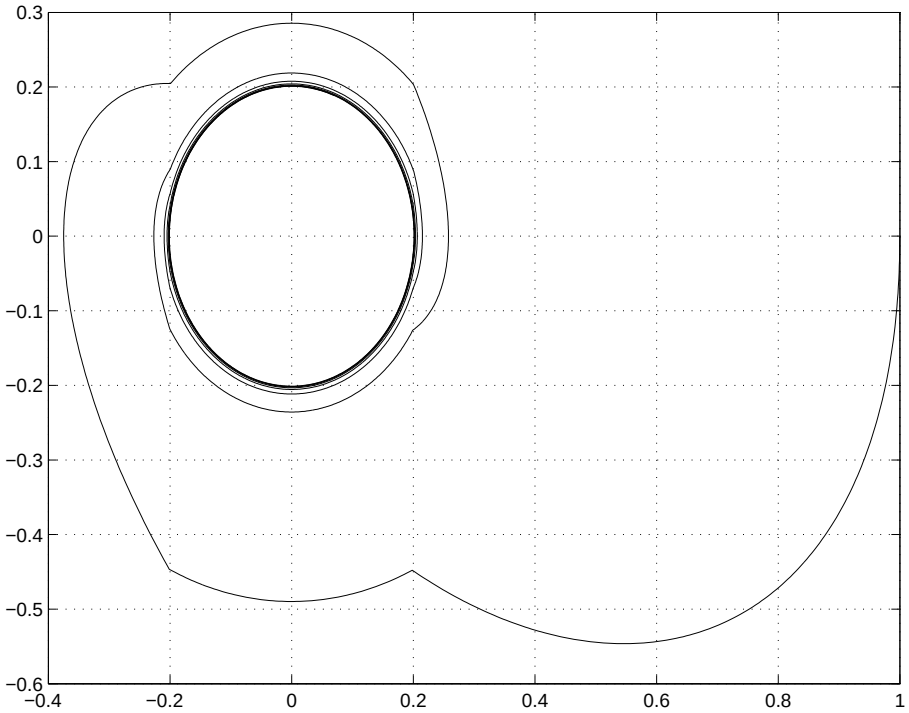


Рис. IX.9. Рис. 18.1 Фазовый портрет управляемой системы

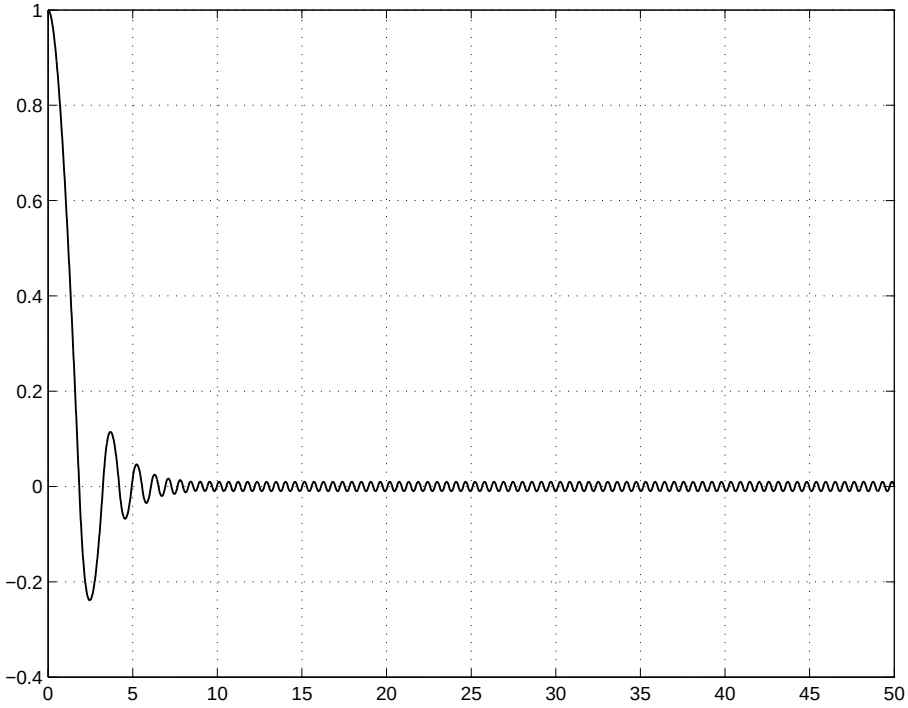


Рис. IX.10. Рис. 18.2 Колебание курса автопилота

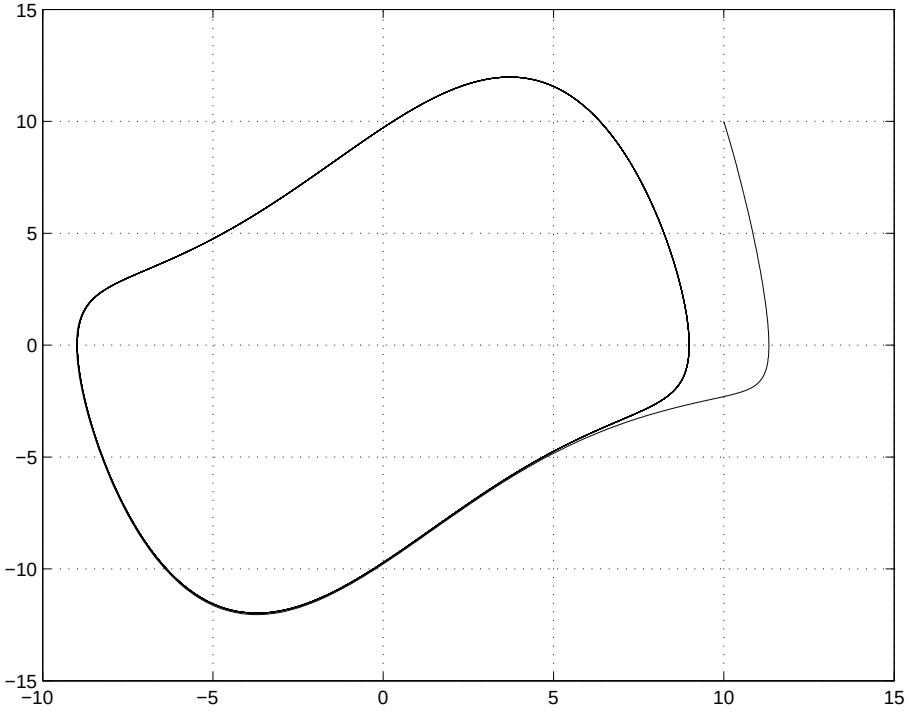


Рис. IX.11. Рис. 19.2 Нелинейный осциллятор Ван-дер-Поля

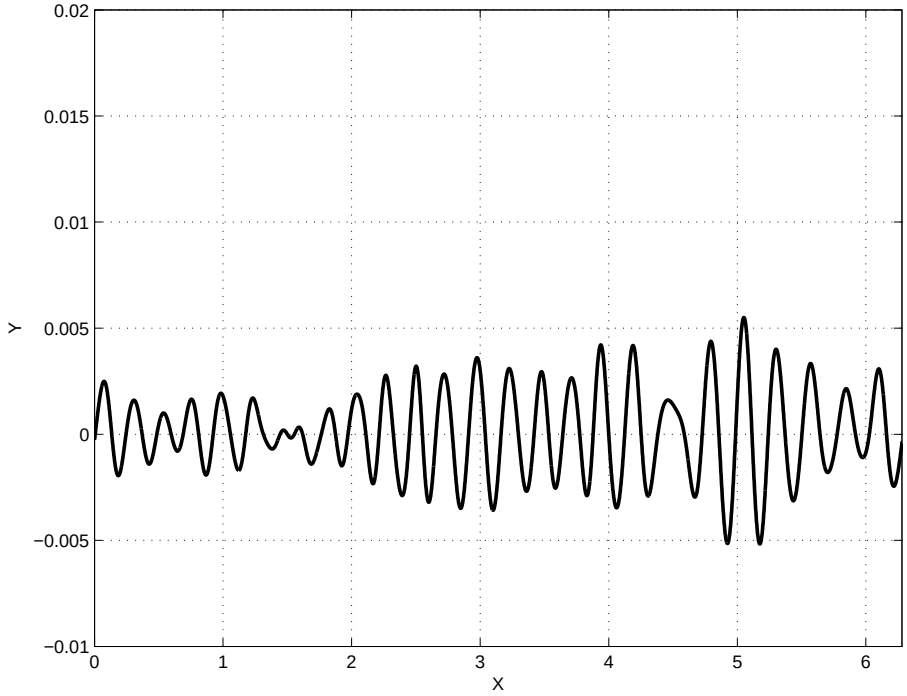


Рис. IX.12. Рис. 22.1 Профиль начальной волны

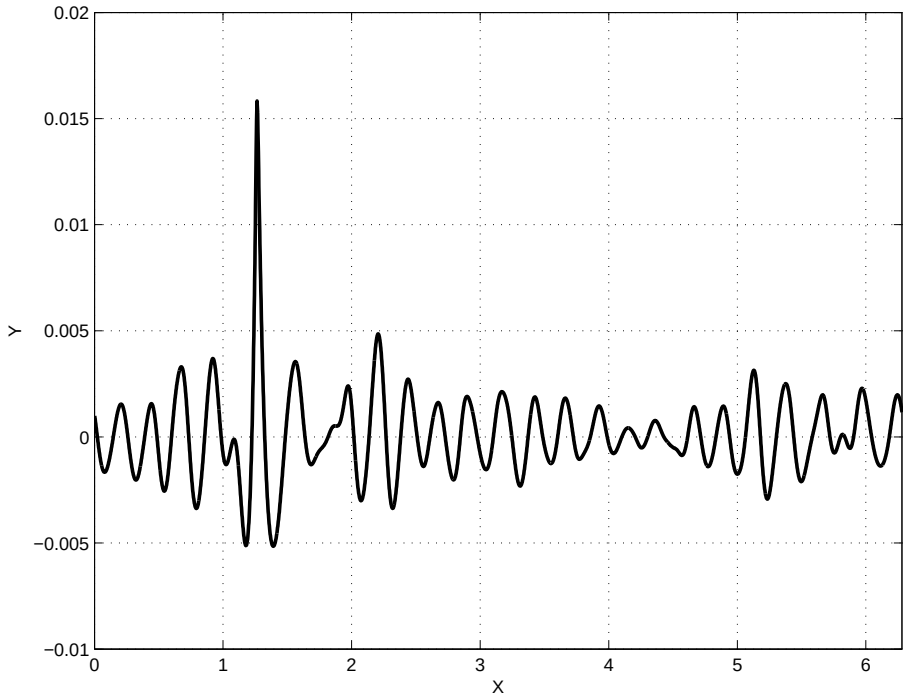


Рис. IX.13. Рис. 22.2 Профиль волны-убийцы

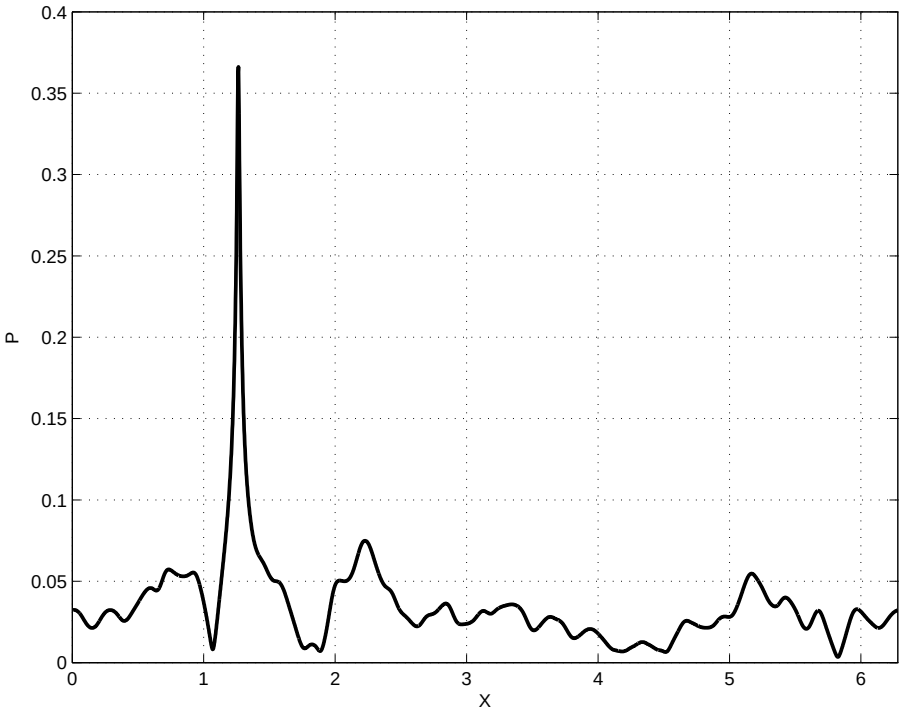


Рис. IX.14. Рис. 22.3 Плотность импульса в момент образования волны-убийцы

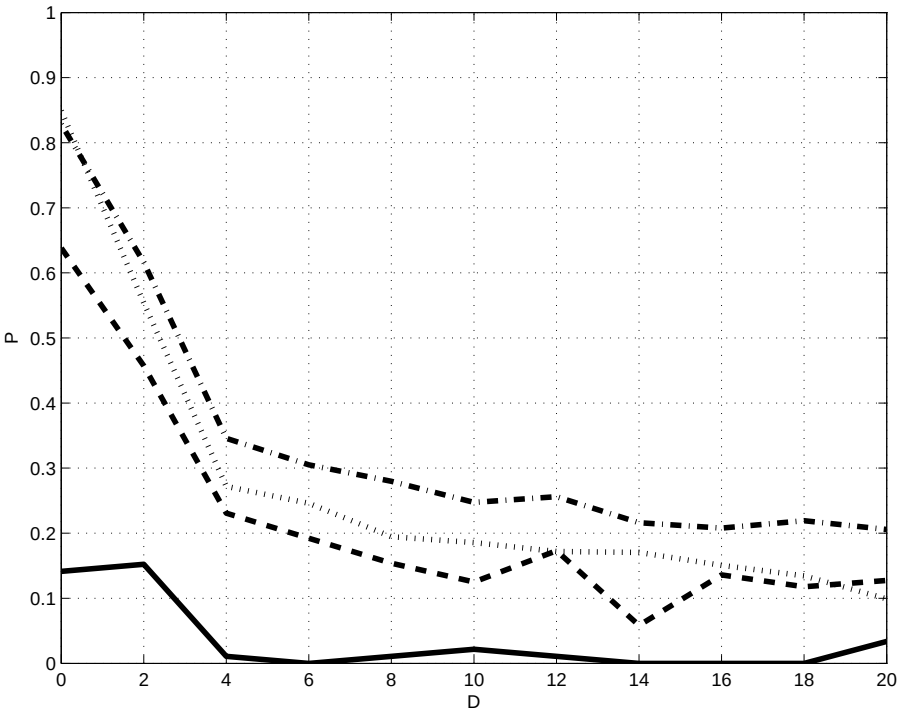


Рис. IX.15. Рис. 22.4 Частоты возникновения волн-убийц