МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»

О.А. АМОСОВА, А.Е. ВЕСТФАЛЬСКИЙ, А.В. КНЯЗЕВ, Н.Е. КРЫМОВ

ЧИСЛЕННЫЕ МЕТОДЫ НА ЯЗЫКЕ PYTHON

Учебное пособие
по курсам
«Численные методы», «Вычислительные методы»

для студентов, обучающихся по направлениям подготовки бакалавров
01.03.02 “Прикладная математика и информатика”,
09.03.01 «Информатика и вычислительная техника»,
12.03.01 «Приборостроение»,
27.03.04 «Управление в технических системах».

Москва Издательство МЭИ 2022

УДК 519
ББК 22.19
Ч-671

Утверждено учебным управлением НИУ «МЭИ»
в качестве учебного пособия

Подготовлено на кафедре математического
и компьютерного моделирования

Рецензенты: докт. физ-мат наук проф. А.В. Перескоков,
ст. преп. Г.В. Крупин

Авторы: Амосова О.А., Вестфальский А.Е., Князев А.В., Крымов Н.Е.
Ч-671 Численные методы на языке Python: учеб. пособие / О.А. Амосова, А.Е. Вестфальский, А.В. Князев, Н.Е. Крымов. – М.: Издательство МЭИ, 2022. – 80 с.

ISBN 978-5-7046-2617-6

Излагаются основы программирования на языке Python. Описываются основные типы данных, операции и выражения языка, управляющие инструкции. Обсуждаются графические возможности языка. Рассматривается работа с файлами. На языке Python реализованы основные алгоритмы, входящие в программу базового курса по численным методам. Дано описание и примеры использования библиотечных функций научных пакетов NumPy и SkiPy.

По курсам «Численные методы», «Вычислительные методы». Образовательная программа «Математическое моделирование».

Для студентов всех образовательныч программ, реализуемых в НИУ «МЭИ» на направлениях подготовки “Прикладная математика и информатика”, «Информатика и вычислительная техника», «Приборостроение», «Управление в технических системах». Для преподавателей и специалистов, занимающихся разработкой программ.

УДК 519
ББК 22.19

ISBN 978-5-7046-2617-6

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Язык Python в настоящее время получил большое распространение при разработке программного обеспечения самого разного назначения.

Язык Python похож на язык C++, но имеет и много отличий.

Язык Python относится к категории объектно-ориентированных языков программирования. В нём практически все данные являются объектами, в том числе, и данные элементарных типов. В переменной хранится ссылка на объект. Ссылки не имеют типов в то время, как сами объекты имеют тип.

Для языка Python разработаны библиотеки, охватывающие обширный круг областей, где используются программы. Имеется несколько мощных пакетов для поддержки математических вычислений, пакеты для распознавания образов и другие.

Не предполагается, что читатели знакомы с языком C++, однако нужно иметь представление о каком-либо языке программирования. Данное пособие предназначено для студентов, обучающихся по различным направлениям, изучающих численные методы.

Пособие состоит из нескольких частей: первая глава содержит сведения об установке и запуске Python, о создании отчетов в среде Jupyter; в главах 2-4 рассмотрены основные элементы языка Python: типы данных, операции, управляющие инструкции, строки, работа с файлами. Обсуждается работа с классами. В главах 5-9 рассмотрены возможности применения языка для решения вычислительных задач. Основное внимание уделено пакетам, используемым в научных вычислениях – NumPY, SciPY. В них реализованы основные алгоритмы линейной алгебры, вычисление интегралов, приближение функций, решение дифференциальных уравнений и систем. Пакет Matplotlib предоставляет хорошие возможности для визуализации данных. В

пособии не представлены символьные вычисления, так как акцент делается именно на вычислительной стороне дела.

Предполагается, что читатель имеет достаточную математическую подготовку, поэтому в главах 5-9 основные сведения по численным методам изложены в сжатом виде. В каждой главе подробно обсуждается решение базовой задачи, затем приводится реализация алгоритма на языке Python и дается решение задачи с помощью библиотечных функций. Авторы надеются, что большое количество примеров поможет читателю справится с практикумом по вычислительной математике в большем или меньшем часовом объеме.

Глава 1. УСТАНОВКА И ЗАПУСК

Python – название языка программирования. Непосредственно для работы вам понадобится установить интерпретатор и стандартную библиотеку (в большинстве Unix систем уже предустановлено), установить пакетный менеджер (pip), а также дополнительные модули для работы и среду разработки.

1.1. Установка

Самым простым способом является установка дистрибутива Anaconda. Он включает все необходимое для работы, а также самые популярные библиотеки для научных вычислений. Для его установки на сайте www.anaconda.com нужно выбрать дистрибутив Python 3 (для требуемой ОС) и следовать инструкциям. В зависимости от архитектуры компьютера и ОС установятся библиотеки, оптимизированные именно для них.

Несмотря на то, что мы рекомендуем устанавливать именно Anaconda (даже для пользователей Unix, где Python уже обычно предустановлен), вы можете скачать последнюю версию Python с официального сайта, либо через пакетный менеджер (для Unix систем). Однако при этом дополнительные модули и библиотеки вам придется устанавливать самостоятельно, а также самостоятельно следить за их обновлениями и совместимостью.

1.2. Запуск и работа

Работать можно непосредственно с интерпретатором в интерактивном режиме. Для Unix систем вы можете запустить его командой ***python*** (***python3***, если ***python*** запускает Python 2) после чего запустится интерактивный режим. Для Windows запуск осуществляется

через консоль набором команды `python` с указанием полного пути к файлу python.exe. Например, если вы установили Anaconda, после запуска консоли можете набрать ***anaconda3\python.exe***. После этого вы увидите в терминале или командной строке примерно следующий текст:

```
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more infor-
mation.
>>>_
```

Теперь можно вводить команды и смотреть результат выполнения.

```
>>> 1+1
2
>>> s="Hello world!"
>>> print(s)
Hello world!
```

Однако это подходит для работы с Python в качестве калькулятора или небольших экспериментов. В реальности код оформляется в виде файлов с расширением «.py» и уже именно эти файлы (скрипты) подаются интерпретатору.

В рамках пособия основным способом работы будет Jupyter.

1.3. Работа в Jupyter

Популярный вариант работы с python – Jupyter. Если вы устанавливали Anaconda, то он был установлен вместе с ним. Запустить его можно через Anaconda-navigator или командой ***jupyter notebook*** в терминале. В Windows он должен появиться в меню «Пуск».

При его запуске откроется терминал/консоль, где запустится специальный сервер, на котором и будут выполняться python-код. После того, как сервер запустился, должен открыться браузер с веб-интерфейсом Jupyter. Если этого не произошло, то взгляните на содержимое терминала/консоли, где можно найти примерно следующие строчки:

```
[I 14:55:54.216 NotebookApp] Jupyter Notebook 6.1.4 is running at:
[I 14:55:54.216 NotebookApp]
http://localhost:8888/?token=c0513dcceb7639796249ec24084656eec5855
dd5
[I 14:55:54.216 NotebookApp]  or
http://127.0.0.1:8888/?token=c0513dcceb7639796249ec24084656eec5855
dd5
```

```
[I 14:55:54.216 NotebookApp] Используйте Control-C, для остановки
этого сервера и выключения всех ядер (дважды, чтобы пропустить
подтверждение).
```

Здесь указано по какому веб-адресу находится сервер и как, если понадобится, завершить его работу. Скопируйте один из этих адресов, например, в нашем случае

```
http://localhost:8888/?token=c0513dcceb7639796249ec24084656eec5855d
d5
```

и вставьте в ваш браузер.

На рисунке 1.1 показан интерфейс программы после запуска. Здесь вы видите файловую структуру компьютера, по которой можете перемещаться и создавать/открывать Jupyter Notebooks.

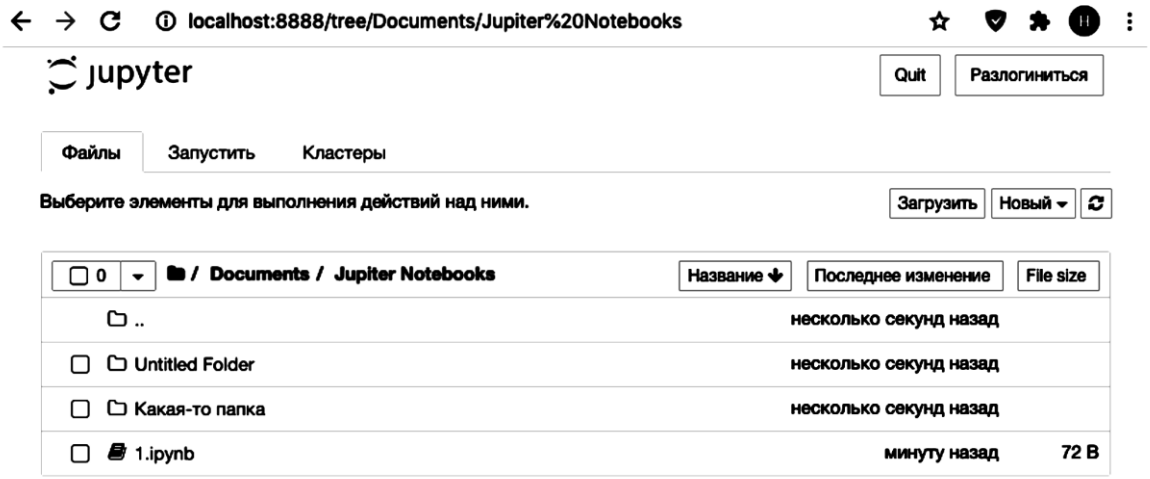

Рис. 1.1

Можно создать новый файл, выбрав ***Новый>Notebook: Python3,*** или открыть существующий файл с расширение **ipynb**.

После создания нового файла или выбора уже существующего (например, на рисунке 1.1 – **1.ipynb**), откроется новая вкладка, в которой будет происходить работа (рис. 1.2).

Рис. 1.2

Jupyter позволяет работать в интерактивном режиме аналогично пакетам Matlab, Mathcad и Mathematica. Разбивая код на ячейки, вы можете вычислить промежуточный результат и отобразить его в текстовом или графическом виде.

Набрав код в ячейке, его можно выполнить, нажав **Запуск** или клавиши **shift+enter.** Область видимости является глобальной, поэтому переменные, определенные в одной ячейке, будут доступны в других ячейках. На рисунке 1.3 в первой ячейке был подключен модуль math, а во второй ячейке была использована функция из этого модуля.

Результат, который был напечатан командой print, а также графики будут находиться под соответствующей ячейкой.

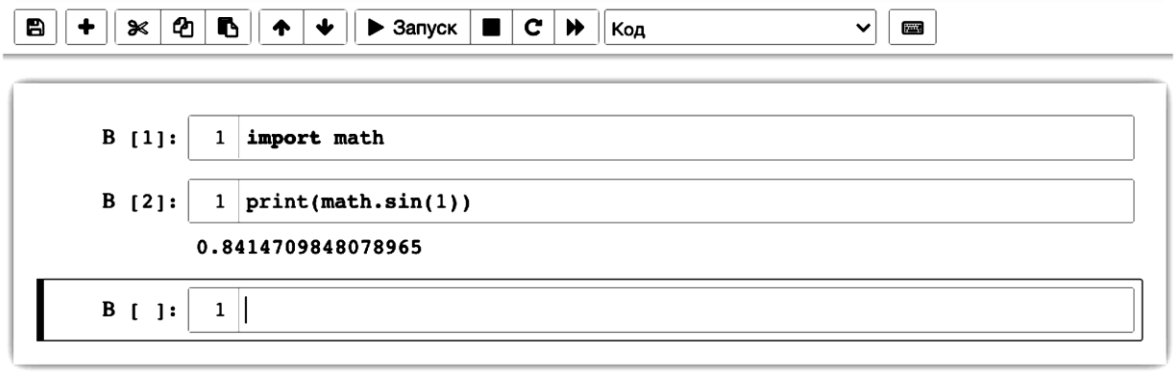

Рис. 1.3

1.4. Оформление отчетов в Jupyter

Ячейка может содержать не только python-код. В меню справа от кнопки **Запуск** есть возможность переключить ячейку в режим Mark-down.

Markdown представляет из себя язык разметки, который может обработать Jupyter. Это позволяет дополнять файл форматированными комментариями.

Разберем на примере основные возможности Markdown. Ниже представлен текст, набранный в ячейке, переведенной в режим форматирования и разметки

```
# Markdown
## Пример 1.1

***

Выше были созданы заголовки 1 и 2 уровня.

---
Текст можно форматировать. Запишем это в виде таблицы:

|    | Вариант 1                | Вариант 2                |
|:---|:-------------------------:|-------------------------:|
|1   | *курсив*                 | _курсив_                 |
|2   | **полужирный**           | __полужирный__           |
|3   | ***полужирный курсив***  | ___полужирный курсив___  |
|4   |                          | ~~зачеркнутый~~          |

Обратите внимание, как с помощью двоеточий были выровнены столбцы.

---

Куда же без списков:
* элемент 1
* элемент 2
    * вложенный элемент 2.1
    * вложенный элемент 2.2
* элемент ...
или
1. элемент 1
2. элемент 2
    1. вложенный
    2. вложенный
3. элемент 3

---

Для вставки кода используются апострофы.

_Команда `import math`  подключает библиотеку **math**._

```
```
```

```
>>>(math.sin(1))
0.8414709848078965
```

```
Имеется поддержка языка верстки LaTeX, с помощью которого можно
набирать простые формулы  $f(x)=e^x$ или более сложные
$$
\int\limits_0^\infty e^{-x^2} = \frac{\sqrt{\pi}}{2}.
$$
```

После набора этого текста можно нажать **Запуск** и получить результат, показанный на рисунке 1.4.

Упомянутый в примере LaTeX выходит за рамки пособия, поэтому предлагаем читателям ознакомиться с ним самостоятельно.

Markdown

Пример 1.1

Выше были созданы заголовки 1 и 2 уровня.

Текст можно форматировать. Запишем это в виде таблицы:

	Вариант 1	Вариант 2
1	*курсив*	*курсив*
2	**полужирный**	**полужирный**
3	***полужирный курсив***	***полужирный курсив***
4		~~зачеркнутый~~

Обратите внимание, как с помощью двоеточий были выравнены столбцы.

Куда же без списков:

- элемент 1
- элемент 2
  - вложенный элемент 2.1
  - вложенный элемент 2.2
- элемент ... или

1. элемент 1
2. элемент 2
   A. вложенный
   B. вложенный
3. элемент 3

Для вставки кода используются апострофы.

Команда `import math` *подключает библиотеку* ***math***.

```
>>>(math.sin(1))
0.8414709848078965
```

Имеется поддержка LaTeX, с помощью которого можно набирать простые формулы $f(x) = e^x$ или более сложные

$$\int_0^\infty e^{-x^2} = \frac{\sqrt{\pi}}{2}.$$

Рис. 1.4

Глава 2. ОСНОВЫ ЯЗЫКА PYTHON

2.1. Введение в Python

2.1.1. Простейшие программы

Напишем самую простую программу:

```
print(“Привет”)
```

Вся программа состоит из одной инструкции.

Ещё одна простая программа:

```
a=17
b=34
c=a+b
print(c)
```

Программа складывает два целых числа и выводит результат на экран. Каждая инструкция располагается в начале строки. Если на строке несколько инструкций, то они должны отделяться точкой с запятой:

```
a=17; b=34; c=a+b
print(c)
```

Можно использовать комментарий:

```
# Это - комментарий
```

2.1.2. Данные

Данные хранятся в виде объектов. Объект или значение имеет тип. Функция ***type(obj)*** возвращает тип объекта ***obj***.

Числа.

К встроенным числовым типам в языке ***Python*** относятся целые числа, числа с плавающей точкой и комплексные числа. Кроме того, встроенная библиотека поддерживает десятичные числа с фиксированной точностью и рациональные дроби.

Целые числа не имеют формальных ограничений по размеру. Они имеют различные формы представления.

Десятичная форма: 121 256732
Двоичная форма: 0b101101
Восьмеричная форма: 0o3751

Шестнадцатиричная форма: 0xA89F6

Числа с плавающей точкой в языке ***Python*** соответствуют числам типа ***double*** в языке ***C***:

0.12 1.34 0.1e-4 12E3

Комплексные числа состоят из двух чисел с плавающей точкой, представляющих действительную и мнимую части комплексного числа:

1.36+0.7j 17.0+0.j 1e0j

Последовательности.

Последовательность – это упорядоченная коллекция элементов, индексируемых с помощью целых чисел.

Встроенные последовательности – это строки, кортежи и списки.

Строка – это последовательность символов:

`*Это - строка*` "*Это – тоже строка*"

В строках могут использоваться *Esc*-последовательности ('\n', '\t' и т.д.).

Кортеж – это не изменяемая упорядоченная последовательность произвольных объектов, типы которых могут различаться:

(*100, 5.0, 200*)
(*1.57*,)
( )

Список – это изменяемая упорядоченная последовательность произвольных объектов, типы которых могут различаться:

[*17, 2.7, "Привет"*]
[*24*]
[ ]

Доступ к элементам последовательностей (строк, кортежей, списков) осуществляется по индексу. Индексация идёт с нуля.

Множества.

Множество – это неупорядоченная коллекция уникальных (неповторяющихся) элементов:

{*18, 3.4, "Строки"*}
{*38*}
set () – пустое множество, нельзя просто {}

Словари.

Отображение – это произвольная коллекция объектов, индексируемых с помощью ключей.

Python предоставляет один встроенный тип отображений: **словарь**. Элементами словаря являются пары "*ключ : значение*":

{`a` : 17, `6` : 36, `c` : 84}
{1:2.0 , 3:17.1 , 24 : 1e+2}
{}

Более подробно кортежи, списки, множества, словари и работа с ними описаны в [3, 4].

Объект None.

Встроенный объект ***None*** означает нулевой объект. Это, по сути дела, пустая ссылка.

Булевы значения.

Булевы значения – это ***True*** и ***False***. Любое значение может интерпретироваться как истинное или ложное (***True*** или ***False***). Любое ненулевое значение выражения или непустой контейнер (кортеж, строка, список и т.д.) в соответствующем контексте рассматривается как истинное значение (***True***). Нуль любого числового типа, пустые контейнеры и ***None*** являются ложными значениями.

2.1.3. Переменные и другие ссылки

Программа на языке ***Python*** получает доступ к значениям данных посредством ссылок. Ссылка – это "имя", которое позволяет обращаться к значению (объекту). Ссылки принимают форму переменных, атрибутов и элементов. В ***Python***`е переменные и другие ссылки не имеют внутреннего типа. Объект, с которым в данный момент связана ссылка, всегда имеет тип, но любая ссылка в процессе выполнения программы может быть связана с объектами разных типов.

Переменные.

В ***Python***`е отсутствуют "объявления" переменных. Существование переменной начинается с того момента, когда впервые встречается инструкция, которая связывает имя переменной с некоторым объектом (иными словами, когда переменная используется для сохранения ссылки на объект).

Можно также освободить переменную, т.е. разорвать ее связь с объектом таким образом, чтобы в ней больше не хранилась никакая

ссылка. Самым распространенным способом связывания переменных и других ссылок является инструкция присваивания. Инструкция ***del*** освобождает переменную, т.е. открепляет ее от объекта.

Возможно повторное связывание ссылки.

Существуют глобальные переменные – это атрибуты объекта модуля. Локальная переменная существует в локальном пространстве имен функции.

Атрибуты и элементы объектов.

Для обозначения атрибута объекта используются ссылки на объект, за которой через точку указывается имя атрибута. Например, ***ob.x*** – атрибут ***x*** объекта ***ob***.

Для обозначения элемента объекта используются ссылки на объект, за которой следует выражение, заключенное в квадратные скобки [ ]. Выражение в скобках – это индекс или ключ элемента, а объект – это контейнер. Например, ***a[i]*** – ***i*** -тый элемент объекта ***a***.

Операции присваивания.

Операции присваивания бывают двух видов: простые и составные. Простые:

цель = выражение

Составные:

цель @ = выражение

Символ @ означает знак некоторой операции (+, -, * и т.д.), например

*a += b * 4*

Значение выражения ***b * 4*** в данном случае прибавляется к ***a***, и результат сложения присваивается переменной ***a***.

В качестве цели могут выступать: идентификатор, ссылка на атрибут, индексированный элемент, срез.

Срез записывается в виде:

объект [начало : конец] *или*
объект [начало : конец : шаг]

Простое присваивание допускает групповые операции, например:

```
a = b = c =5
```

выражение справа вычисляется один раз, и затем ссылка на этот объект присваивается переменным слева направо.

Возможно присваивание с распаковкой, например:

```
a, b, c  =  x
```

Здесь предполагается, что ***x*** – это последовательность, содержащая 3 элемента. Такое присваивание эквивалентно:

```
a = x [0]
b = x [1]
c = x [2]
```

Составные присваивания:

```
+ =
- =
* =
```
… и т.д.

Инструкция del.

Инструкция ***del*** не удаляет объекты в буквальном смысле, а лишь открепляет ссылки, т.е. разрывает связь между именем и объектом, с которым ассоциировано данное имя.

Пример

```
del  x, y, z
```

Объекты, на которые отсутствуют ссылки, удаляются специальной программой – сборщиком “мусора”, работающей в фоновом режиме.

2.1.4. Операции

Арифметические операции.

Используются следующие арифметические операции:

+ – сложение
- – вычитание
* – умножение
/ – деление, результат всегда вещественное число
// – деление, результат – целое, преобразованное к более широкому

типу, дробная часть отбрасывается, например:

7 // 5 результат 1
7.0 // 5 результат 1.0
-7 // 2 результат -4

% – остаток от деления
** – возведение в степень
- – унарный минус
+ – унарный плюс

Логические операции.
Используются следующие логические операции:

not – логическое НЕ
and – логическое И
or – логическое ИЛИ

При выполнении логических операций И и ИЛИ второй операнд не будет вычисляться, если результат полностью определяется первым операндом.

Поразрядные операции.
Поразрядные операции выполняются над целыми числами:

~ – поразрядная инверсия
& – поразрядная конъюнкция
| – поразрядная дизъюнкция
^ – поразрядное исключающее ИЛИ
<< – сдвиг влево
>> – сдвиг вправо (самый левый разряд заполняется знаком)

Операции сравнения.
Это – обычные операции сравнения:

= = != > >= < <=

Возможны цепочки сравнений:

a > b >= c > d

Это – то же самое, что и:

a > b and b >= c and c > d

Операции над последовательностями.

Здесь будут рассмотрены операции, являющиеся общими для всех видов последовательностей. Специфические операции будут рассмотрены позднее.

\+ – конкатенация
* – повторение

Если ***S*** – последовательность, то ***S * n*** или ***n * S*** – это ***n*** раз выполненная конкатенация.

Операция проверка на принадлежность:

x in S – результат ***True*** или ***False***

Операция индексирования позволяет получить доступ к элементу последовательности:

S – последовательность
S [i] – элемент последовательности

Индексы находятся в диапазоне $0 \div (\boldsymbol{n} - 1)$, где ***n*** – число элементов в последовательности. При этом:

S [-1] то же самое, что *S [**n** - 1]*
S [-2] то же самое, что *S [**n** - 2]*
…
*S [-**n**]* то же самое, что *S [0]*

При работе с последовательностями производится проверка индексов.

Операция взятия среза:

S [i : j] – включает элементы *S [i], S [i+1] … S [j-1]*

Возможен срез с шагом:

S [i : j : k] – включает элементы *S [i], S [i+κ] …*

2.1.5. Простейший ввод-вывод

Для вывода на экран используется функция ***print***:

print (<перечень объектов>)

Например:

```
a = 12
c = (3,4)
s = “Строка 1”
print(a,c,s)
```

Для ввода с клавиатуры используется функция ***input***:

[<значение>=] input ([<сообщение>])

Функция возвращает вводимую строку. При этом можно вывести сообщение, обычно используемое как приглашение к вводу.

2.2. Управляющие инструкции

2.2.1. Инструкция if

Структура инструкции:

if *<выражение>* **:**
    <инструкции>
elif *<выражение>* **:**
    <инструкции>
elif *<выражение>* **:**
    <инструкции>
. . .
e*lse* **:**
    <инструкции>

Предложения ***elif*** и ***else*** могут отсутствовать.

В языке нет операторных скобок для обозначения составной инструкции. Роль этих скобок выполняют отступы. Обычно используются 4 пробела в качестве отступа.

В языке ***Python*** нет инструкции ***switch***.

Пример использования инструкции ***if***:

```
x = int(input(“Введите целое число: ”))
```

```
if x < 0 :
    y = x
    print(“x – отрицательное число”)
elif x > 0 :
    z = x
    print(“x – положительное число”)
else :
    print(“x равно нулю”)
```

Пользователь набирает на клавиатуре целое число, которое вводится как строка и затем преобразуется в целое.

2.2.2. Инструкция while

Структура инструкции:

while <*выражение*> **:**
<*инструкции*>

Инструкции в теле цикла выполняются, пока *выражение* истинно.

Например:

```
i = 10
while i :
    print(i)
    i -= 1
```

Цикл выполняется, пока ***i*** не станет равным нулю.

2.2.3. Инструкция for

Структура инструкции:

for <*цель*> *in* <*итерируемое выражение*> **:**
<*инструкции*>

Как правило, цель – это идентификатор, который представляет управляющую переменную цикла. Итерируемое выражение – это, в частности, последовательность. Управляющая переменная получает очередное значение из последовательности, и для этого значения выполняются инструкции. Цикл заканчивается, когда исчерпывается последовательность.

Рассмотрим несколько примеров.

Пример 1. Вычислим сумму положительных элементов списка (массива – массивов как таковых в чистом языке ***Python*** нет, они представляются обычно списками).

```
a = [2,-1,3,-4,0,5,2]
s = 0
for x in a :
    if x > 0 :
        s += x
print(s)
```

Пример 2. Подсчитаем по отдельности количество положительных и отрицательных элементов в списке.

```
b = [-1,4,-2,0,3,1,0,-1,2]
kpos = 0
kneg = 0
for x in b :
    if x > 0 :
        kpos += 1
    elif x < 0 :
        kneg += 1
print(kpos,kneg)
```

Функция range.

Функция ***range*** генерирует последовательность целых чисел. Имеется несколько форм этой функции:

range(n) – генерируется последовательность чисел от 0 до ***n – 1***
range(m,n) – генерируется последовательность чисел от ***m*** до ***n – 1***
range(m,n,h) – генерируется последовательность чисел от ***m*** до ***n – 1*** с шагом ***h***.

Пример 3.

```
n = 20
for i in range(n):
    print(i)
```

Здесь выводится последовательность чисел 0, 1, 2, … , 19. Заголовок цикла в данном случае эквивалентен записи `for(i=0;i<n;i++)` в языках ***C++, C#, Java***.

Функция len.

Функция ***len*** определяет количество элементов в последовательности.

Пример 4. Увеличим положительные элементы списка вдвое, а к отрицательным элементам прибавим число 3.

```
a = [3,1,-2,0,-1,2,1,-4,0,-7]
for i in range(len(a)):
    if a[i]>0:
        a[i]*=2
    elif a[i]<0:
        a[i]+=3
print(a)
```

Если нужно было бы провести эти изменения только для элементов с чётными индексами, то заголовок цикла имел бы вид:

```
for i in range(0,len(a),2):
```

Генераторы списков.

Генератор списка – это специальное выражение, которое обеспечивает формирование нового списка, используя заданный список. Структура генератора списков:

[<выражение> for <цель> in <последовательность> <предложение>]

Здесь *предложение* имеет одну из следующих форм:

for <цель> in < последовательность >
if <выражение>

В генераторе может быть несколько предложений.

Пример 5. Используем генератор для формирования нового списка.

```
a = [2,-3,0,1,4,-2]
b = [x for x in a if x>0]
print(a)
```

Здесь формируется список ***b*** из положительных элементов списка ***a***.

Существуют аналогичные генераторы для кортежей и множеств, но выражения должны быть заключены в круглые или фигурные скобки соответственно.

2.2.4. Инструкции break и continue

Эти инструкции работают так же, как и в ***C++, C#*** или ***Java***. Структура инструкций:

break
continue

Напишем программу подсчёта числа элементов списка, находящихся до первого отрицательного элемента.

```
a = [4,2,0,1,3,0,-2,3,4,-1]
kol = 0
for x in a:
    if x<0:
        break
    kol += 1
print(kol)
```

Когда встречается первый отрицательный элемент, цикл прерывается.

2.2.5. Предложение else в инструкциях циклов

Инструкции ***while*** и ***for*** могут включать необязательное предложение ***else***, например:

for . . .
. . .
else:
    <инструкции>

Предложение ***else*** выполняется, если цикл завершается естественным образом, а не путём прерывания с помощью ***break*** или ***continue***.

Напишем программу вывода элементов списка до первого отрицательного элемента.

```
a = [4,2,0,1,3,-2,1,4,-3]
for x in a:
    if x<0:
        break
    else:
        print(x)
```

```
else:
    print(“В списке нет отрицательных значений”)
```

Если цикл дошёл до конца, и отрицательные элементы не встретились, то выводится сообщение об отсутствии отрицательных значений.

2.2.6. Инструкция pass

Иногда синтаксис требует наличия инструкции, а она там не нужна. В этом случае можно использовать пустую инструкцию ***pass***, например:

```
x = int(input(“Введите значение x: ”))
if x>4:
    pass
else:
    print(x)
```

Здесь выводятся значения введённой переменной, не превышающие величины 4.

2.3. Функции

2.3.1. Определение функции

Структура определения функции имеет следующий вид:

def <имя функции> (<список параметров>):
    <инструкции>

Если тело функции не содержит инструкций, то должен быть указан оператор ***pass***:

def fun():
  pass

В теле функции может быть оператор ***return***, возвращающий результат.

Пример.

```
def Sqr2(x):
    y=x**2
    return y
a=3
b=Sqr2(a)
c=2.5
d=Sqr2(c)
g=1+2j
f=Sqr2(g)
print(b)
print(d)
print(f)
```

С помощью функции ***Sqr2*** в квадрат возводится целое число, число с плавающей точкой и комплексное число. Результаты работы программы:

9
6.25
(-3+4j)

Заметим, что функция всегда возвращает значение. Если нет явного возвращения результата с помощью оператора ***return***, то возвращается значение ***None***.

```
def fun1(x,y):
    z=x+y
    print(z)
fun1(3,4)
d=fun1(12,13)
print(d)
```

Результат работы программы:

7
25
None

2.3.2. Параметры функции

При вызове функции фактические аргументы передаются по значению, т.е. они копируются в параметры функции, которые являются локальными переменными в пространстве имён функции. Напомним, что имена переменных, в том числе и фактические аргументы и формальные параметры функции являются ссылками.

В функции могут использоваться параметры нескольких видов. В начале списка параметров должны размещаться позиционные параметры (или обязательные параметры). Они задаются простыми идентификаторами.

Рассмотрим сначала позиционные параметры.

Пример 1. Напишем функцию для вычисления суммы и количества элементов массива, больших заданной величины. Для представления массива будем использовать список:

```
def fun2(x,n,a):
    s=0
    k=0
    for i in range(n):
        if x[i]>a:
            s+=x[i]
            k+=1
    y=(s,k)
    return y
a=[3,-7,4,0,2,-3,1]
y=fun2(a,7,0)
print(y)
z=fun2(a,7,1)
print(z)
```

Для возврата результата функции используется кортеж. Результат работы программы:

(10,4)
(9,3)

Хотя значения передаются в функцию по ссылке, неизменяемые объекты (в том числе и числа) внутри функции изменить нельзя.

Пример 2.

```
def fun3(x,y):
    x=33
    y[0]=25
    y.append(7)
a=4
b=[1,2]
fun3(a,b)
print(a)
print(b)
```

Результат работы программы:

4
[25,2,7]

Список действительно изменился, а переменная ***a*** – нет.

Вслед за списком позиционных параметров может располагаться список именованных параметров (необязательных параметров). Каждый элемент этого списка имеет вид:

<идентификатор> = <выражение>

Значение выражения – это значение по умолчанию, которое передаётся параметру, если соответствующий фактический аргумент не задан.

Пример 3.

```
def fun4(x,n,a=0):
    s=0
    k=0
    for i in range(n):
        if x[i]>a:
            s+=x[i]
            k+=1
    y=(s,k)
    return y
a=[3.1,-7.2,4.4,0,2.3,-3.4,1.1]
y=fun4(a,7)
print(y)
z=fun4(a,7,1.2)
print(z)
```

Здесь используется именованный параметр ***a***. Если он не задаётся (как при первом вызове функции), то в функцию передаётся значение по умолчанию. Результат работы программы:

(10.9,4)
(9.8,3)

В отличие от ***C++*** именованные параметры могут опускаться не только в конце.

Пример 4.

```
def fun5(a,b=1,c=2,d=3):
```

```
    y=a+b+c+d
    print(y)
fun5(2)
fun5(2,3)
fun5(2,c=4)
fun5(2,d=7)
```

Результат работы программы:

8
10
10
12

В конце списка параметров можно использовать любую из двух специальных форм ****args*** или *****kwds*** (или обе вместе).

Форма ****args*** означает, что дополнительные позиционные аргументы объединяются в кортеж и связываются с параметром ***args***.

Пример 5.

```
def fun6(a,b=3,*c):
    prod=a*b
    for x in c:
        prod*=x
    return prod
y=fun6(4,1,2,3)
print(y)
```

Аргументы 2 и 3 объединяются в кортеж и связываются с параметром функции ***c***. Результат работы программы:

24

Форма *****kwds*** означает, что дополнительные именованные аргументы объединяются в словарь и связываются с параметром ***kwds***.

Пример 6.

```
def fun7(a,b=3,**d):
    s=a+b
    print(d)
    return s
z=fun7(4,b=1,i=2,j=3)
print(z)
```

Фактические аргументы (два последних) преобразуются в словарь. Результаты работы программы:

{'i':2,'j':3}
5

Между аргументами ***args*** и *****kwds*** могут размещаться именованные параметры, и при этом соответствующие аргументы должны передаваться только по именам.

Переменные внутри функции скрывают глобальные переменные, связанные вне функции. Если нужно сделать видимыми в некоторой функции глобальные переменные, то первой инструкцией в функции должна быть инструкция

global <идентификаторы>

В качестве параметров функции может выступать другая функция.

Пример 7. Напишем программу, которая подсчитывает сумму квадратов и сумму кубов элементов списка (по отдельности).

```
def fun8(f,a):
    s=0
    for x in a:
        s+=f(x)
    return s
def f1(x):
    return x**2
def f2(x):
    return x**3
b=[2,-1,3,0,4]
s1=fun8(f1,b)
s2=fun8(f2,b)
print(s1,s2)
```

Результатом работы программы будет сумма квадратов и сумма кубов элементов списка ***b***.

2.3.3. Вложенные функции

Одна функция может быть определена внутри другой функции. Вложенная функция имеет доступ к переменным, определённым во внешней функции. Далее приведён пример вложенной функции:

```
def func():
    a=5
    def fun1(x):
        y=a+x
        return y
    b=fun1(6)
    print(b)
func()
```

Функция ***fun1*** определена внутри функции ***func***. Она имеет доступ к переменной ***a***, определённой в функции ***func***.

2.3.4. Рекурсивные функции

Функция может обращаться сама к себе, т.е. быть рекурсивной. Напишем функцию, вычисляющую факториал. В библиотеке Python есть такая функция – мы пишем её просто для иллюстрации.

```
def fact(n):
    if n>0:
        return n*fact(n-1)
    else:
        return 1
a=fact(5)
print(a)
b=fact(6)
print(b)
```

В результате будет выведено:

120
720

2.3.5. Анонимные функций

В языке ***Python*** можно использовать лямбда-функции, часто называемые также анонимными функциями, т.к. они не имеют своего специфического имени.

Структура лямбда-функции:

lambda [<список параметров>]:<выражение>

Функция вычисляет и возвращает указанное выражение.

В качестве значения анонимная функция возвращает ссылку на объект-функцию, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызывается анонимная функция, как и обычная, с помощью круглых скобок, в которых передаются параметры.

Пример:

```
f=lambda x,y: x*y+1
a=f(2,3)
print(a)
```

В результате будет выведено значение 7.

Рассмотрим теперь пример передачи объекта-функции в качестве параметра в другую функцию:

```
def fun1(f,x):
   y=f(x)+2
   return y
f2=lambda x:x**4
z=fun1(f2,3)
print(z)
```

В результате будет выведено значение 83.

Анонимную функцию можно определить сразу в списке параметров другой функции:

```
def fun1(f,x):
   y=f(x)+2
   return y
z=fun1(lambda x: 5*x,3)
print(z)
```

Будет выведен следующий результат: 17.

2.4. Модули

Модуль – это любой файл с программным кодом. Каждый модуль может импортировать другой модуль, получая таким образом доступ к атрибутам этого модуля (переменным, функциям и классам, объявленным внутри импортируемого модуля).

Программы, которые запускались ранее, находились в модуле ***__main__***.

Для загрузки модуля используется инструкция ***import***. Её структура:

import <имя модуля 1> [as <псевдоним>] [,<имя модуля 2>
[as <псевдоним>] …]

Например:

```
import math
import NumPy as np
```

Если нужно импортировать не все атрибуты из некоторого модуля, то можно использовать инструкцию ***from***. Её структура:

from <имя модуля> import <имя атрибута>
[as <имя переменной>][, …]

Инструкция

*from <имя модуля> **

позволяет импортировать все атрибуты модуля.

В качестве примера рассмотрим модуль ***math***, который содержит математические функции и константы.

Инструкция импорта:

```
import math
```

После этого можно использовать атрибуты модуля:

```
math.pi
math.e
math.sin(x)
math.cos(x)
math.tan(x)
math.asin(x)
math.acos(x)
math.atan(x)
math.exp(x)
math.log(x)
math.tan(x)
math.log10(x)
math.log2(x)
math.sqrt(x)
math.fabs(x)
```

Пакет – это средство группирования модулей. При создании пакета создаётся каталог с соответствующим именем, в котором размещаются модули и подкаталоги (подпакеты).

2.5. Работа с файлами

2.5.1. Чтение и запись файлов

Для открытия файла используется функция ***open***. Её формат:

open <путь к файлу> [,<режим>='r']

Первый параметр – это строка, представляющая полный путь к файлу, включая имя файла. Второй параметр – это режим открытия файла (как в языке ***C***). Функция возвращает ссылку на объект файла. Файл может быть открыт в текстовом или двоичном режимах.

После работы файл должен быть закрыт методом ***close***:

close()

Для записи в файл используется метод ***write***:

write(<данные>)

В текстовом режиме метод записывает в файл строку, в двоичном – последовательность байтов.

Для считывания из файла используется метод ***read***:

read([<количество>])

Если параметр (число) указан, то считывается заданное количество символов или байтов. Если параметр не указан, то в текстовом режиме возвращается строка, а в двоичном режиме – содержимое файла от текущей позиции до конца файла.

Пример.

```
f=open("File1.txt","wt")
f.write("Строка1\n")
f.write("Строка2\n")
f.write("Строка3")
f.close()
```

Здесь в файл записываются три строки. Теперь считаем эти строки из файла и выведем на экран:

```
f=open("File1.txt","rt")
s1=f.read()
s2=f.read()
s3=f.read()
f.close()
print(s1)
print(s2)
print(s3)
```

Будет выведено:

Строка1
Строка2
Строка3

Метод ***readlines*** всё содержимое текстового файла в список строк:

```
f=open("File1.txt","rt")
ls=f.readlines()
f.close()
print(ls)
```

Будет выведено:

['Строка1\n','Строка2\n','Строка3\n']

После получения списка можно обратиться по индексу к любой строке.

Если нужно просто последовательно обработать строки текстового файла, можно использовать инструкцию ***for***:

```
f=open("File1.txt","rt")
for s in f:
    print(s)  # Здесь может быть любая обработка строки
f.close()
```

2.5.2. Сериализация

Можно сохранить объекты в файле, потом считать их из файла. Этот механизм часто называется сериализация/десериализация.

Модуль ***pickle*** содержит метод ***dump***, который записывает объект в файл, и метод ***load***, который позволяет считать объект из файла.

Структура этих методов имеет следующий вид:

dump(<объект>,<файл>)
load(<файл>)

Метод ***load*** возвращает ссылку на считанный объект. Файл должен быть открыт в двоичном режиме.

Пример.

Запишем в файл некоторые объекты:

```
import pickle
a=15
b=(1,2)
c=["Строка",4,[7,9]]
f=open("File3.bin","wb")
pickle.dump(a,f)
pickle.dump(b,f)
pickle.dump(c,f)
f.close()
```

Теперь считаем объекты из файла:

```
import pickle
f=open("File2.bin","rb")
a=pickle.load(f)
b=pickle.load(f)
c=pickle.load(f)
f.close()
print(a)
print(b)
print(c)
```

На выходе получим:

15
(1,2)
['Строка',4,[7,9]]

Глава 3. МОДУЛЬ NUMPY

Основным объектом модуля ***NumPy*** является однородный многомерный массив ***numpy.ndarray***. Это – многомерный массив элементов (обычно чисел) одного типа.

Наиболее важные атрибуты объектов ***ndarray***:

ndarray.ndim – число измерений (осей) массива
ndarray.shape – кортеж, задающий размеры массива по каждой из осей
ndarray.dtype – тип элементов массива

Имеется ряд встроенных типов элементов массива: ***bool_, character, int8, int16, int32, int64, float8, float16, float32, float64, object_***. По умолчанию задаётся тип ***float64***.

3.1. Создание массивов

Существует несколько способов для создания массива.

Функция array().
Её простой заголовок имеет вид:

array(<последовательность>,dtype=None)

Возвращается соответствующий массив. Тип элементов зависит от типа элементов исходной последовательности, но его можно переопределить при создании массива:

```
import numpy as np
a = np.array([2,1,4,-5])
print(a)
```

Будет выведено:

array([2,1,4,-5])

При создании можно указать тип элементов массива:

```
import numpy as np
a = np.array([[1,2,3],[-1,-2,-3]],dtype=np.float32))
print(a)
```

Будет выведено:
array([[1.,2.,3.],

[-1.,-2.,-3.]])

Функции zeros(),ones().

Функция ***zeros()*** создаёт массив из нулей, функция ***ones()*** создаёт массив из единиц.

Структура этих функций имеет вид:

zeros(<кортеж с размерами>,dtype=None)
ones(<кортеж с размерами>,dtype=None)

Функция eye().

Эта функция создаёт единичную матрицу, например:

```
. . .
a = np.eye(4)
print(a)
```

Будет выведено:

array([1.,0.,0.,0],
[0.,1.,0.,0],
[0.,0.,1.,0],
0.,0.,0.,1])

Функция empty().

Функция создаёт массив без его заполнения (в нём “мусор”).

Функция arrange().

Функция создаёт одномерный массив, содержащий последовательность значений от начального значения до конечного с заданным шагом, например:

```
. . .
b = np.arrange(0,2,0.4)
print(b)
```

Будет выведено:

[0.,0.4,0.8,1.2,1.6,1.9999]

Функция linspace().

Функция создаёт массив с заданным количеством элементов:

```
. . .
b = np.linspace(0,2,9)
print(b)
```

Будет выведено:

[0.,0.25,0.5,0.75,1.,1.25,1.5,1.75,2.]

Функция fromfunction().

Функция создаёт массив, применяя параметр-функцию ко всем комбинациям индексов:

```
. . .
def fun(i,j):
   return 10*i+j
. . .
a = np.fromfunction(fun,(3,4))
print(a)
```

Будет выведено:

[[0., 1., 2., 3.]
[10.,11., 12., 13.]
[20.,21., 22., 23.]]

3.2. Операции над массивами

Математические операции над массивами выполняются поэлементно:

```
. . .
a = np.array([10,20,30,40])
b = np.array([0,1,2,3])
c = a + b
print(c)
```

Будет выведено:

[10,21,32,43]

Следующий пример:

```
. . .
d = a * b
print(d)
```

Будет выведено:

[0,20,60,120]

Ещё пример:

```
. . .
```

```
f = a ** b
print(f)
```

Будет выведено:

[1,20,900,64000]

Возможны операции над массивом и числом:

```
. . .
a = np.array([10,20,30,40])
b = a + 3
print(b)
```

Будет выведено:

[13,23,33,43]

В пакете ***NumPy*** определён ряд функций (***sin, cos*** и т.д.) для поэлементных вычислений с массивами:

```
. . .
x = np.array([0.,0.5,1.0,1.5])
y = np.sin(x)
print(y)
```

Будет выведено:

[0.,0.479 . . .,0.841 . . .,0.997 . . .]

Имеются методы для вычисления суммы всех элементов массива, поиска наименьшего и наибольшего элементов массива:

```
. . .
a = np.array([3,-4,0,5,2,1])
s = a.sum()
print(s)
c = a.min()
print(c)
d = a.max()
print(d)
```

Будет выведено:

7
-4
5

По умолчанию эти операции применяются к массиву в целом. Однако, если указать параметр ***axis***, можно использовать операцию для заданной оси массива:

```
. . .
a = np.array([[3,-4,0,5],[0,1,2,3],[1,-1,2,0]])
b = a.min(axis=0)  # наименьшее значение в каждом столбце
print(b)
c = a.min(axis=1)  # наименьшее значение в каждой строке
print(c)
```

Будет выведено:

[0, -4, 0, 0]
[-4, 0, -1]

Можно использовать срезы, например:

```
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
print(a[0, 1:3])
print(a[:, 1:3])
print(a[0::2, ::-1])
```

Будет выведено:

[2 3]
[[2 3]
[5 6]
[8 9]]
[[3 2 1]
[9 8 7]]

Глава 4. ГРАФИКА

4.1. Библиотека Matplotlib

Графика в программах на языке ***Python*** поддерживается различными средствами. В данном пособии рассматривается библиотека ***matplotlib***. В её состав входит ряд модулей, в том числе, модуль высокоуровневой графики ***pyplot***.

Инструкция включения этого модуля в программу обычно выглядит следующим образом:

import matplotlib.pyplot as plt

Изображение в ***matplotlib*** создаётся путём последовательного вызова команд/функций этого модуля. Графические объекты (точки, линии, фигуры и т.д.) последовательно накладываются один на другой, если они занимают общие области на рисунке.

Объектом самого высокого уровня при работе с ***matplotlib*** является рисунок (***Figure***). На нём располагаются одна или несколько областей рисования (***Axes***) и элементы оформления рисунка (заголовки, легенда и т.д.). Каждый объект ***Axes*** содержит две (или три) координатных оси ***Axis***. Но основное назначение объекта ***Axes*** состоит в том, что на него наносится графика: кривые, диаграммы и так далее.

Создать рисунок ***Figure*** позволяет инструкция:

```
plt.figure()
```

Объект ***Figure*** может автоматически создаваться при первом выполнении какой-либо графической функции.

4.2. Линейные графики

Для рисования графиков используется функция ***plot*** модуля ***pyplot***. Один из вариантов функции ***plot***:

*plot([x,]y,[fmt,]**kwargs)*

Здесь ***x, y*** – списки с абсциссами и ординатами точек соответственно, ***fmt*** – базовое форматирование (цвет, стиль линии), ***kwargs*** – набор именованных параметров.

Если список ***x*** отсутствует, то предполагается, что абсциссы принимают значения: 0, 1, 2, … .

Функция ***plot*** создаёт график как объект. Для его визуализации используется метод ***show*** модуля ***pyplot***.

Рассмотрим следующий пример:

```
import matplotlib.pyplot as plt
x = [0.0,0.2,0.4,0.6,0.8,1.0]
y = [1.2,1.7,2.0,2.1,1.9,1.8]
plt.plot(x,y)
plt.show()
```

В результате будет нарисована соответствующая линия, как показано на рис. 4.1:

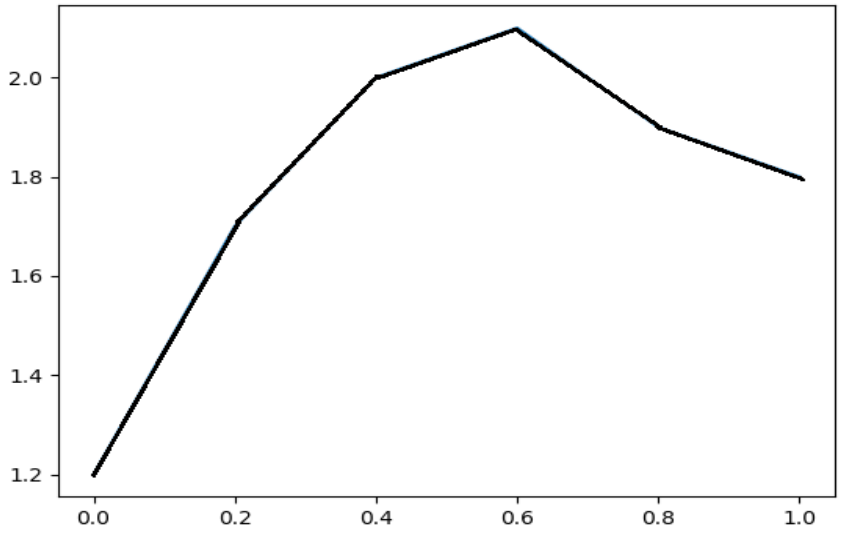

Рис. 4.1

Параметр ***fmt*** имеет следующую структуру:

‘[<маркер>][<стиль линии>][<цвет линии>]’

Некоторые виды маркеров:

‘o’ – небольшая окружность
‘v’ – небольшой треугольник (вершиной вниз)
‘^’ – небольшой треугольник (вершиной вверх)
‘s’ – небольшой квадрат
‘p’ – небольшой пятиугольник
‘’* – звездочка (*)
‘+’ – знак +

Некоторые виды линий:

‘–’ – сплошная линия
‘– –’ – штриховая линия
‘– .’ – штрих-пунктирная линия
‘:’ – пунктирная линия

Некоторые цвета:

‘b’ – синий
‘g’ – зелёный
‘r’ – красный
‘c’ – бирюзовый (cyan)
‘m’ – вишнёвый (magenta)
‘y’ – жёлтый
‘k’ – чёрный
‘w’ – белый

Если в строке форматирования используется только цвет, то его название можно писать полностью, например ***'green'***, или в шестнадцатиричном коде – ***'#00FF00'*** (используется система ***RGB***).

Параметр *****kwargs*** даёт возможность задавать именованные параметры, например:

linewidth=3
linestyle='dashed' (или 'solid', 'dashdot', 'dotted')
markersize=10
color='green'
marker='o'

Рассмотрим следующий пример (задаётся цвет линии и маркер):

```
import matplotlib.pyplot as plt
x = [0.0,0.2,0.4,0.6,0.8,1.0]
y = [1.2,1.7,2.0,2.1,1.9,1.8]
plt.plot(x,y,'or-')
plt.show()
```

Линия и маркеры здесь стали красными.

В следующем примере используется именованный параметр для задания толщины линии:

```
import matplotlib.pyplot as plt
x = [0.0,0.2,0.4,0.6,0.8,1.0]
y = [1.2,1.7,2.0,2.1,1.9,1.8]
plt.plot(x,y,'or-',linewidth=5)
plt.show()
```

Линия на графике стала толстой.

Асбциссы точек удобнее задавать как отрезок, делённый на заданное количество частей. Для этого используется функция ***linspace*** модуля ***NumPy*** (см выше). В следующем примере абсциссы задаются с помощью этой функции:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0,2,11)
y = [1.2,1.7,2.0,2.1,1.9,1.8,1.6,1.3,1.0,0.8,0.7]
plt.plot(x,y)
plt.show()
```

В результате получим следующую кривую (рис. 4.2):

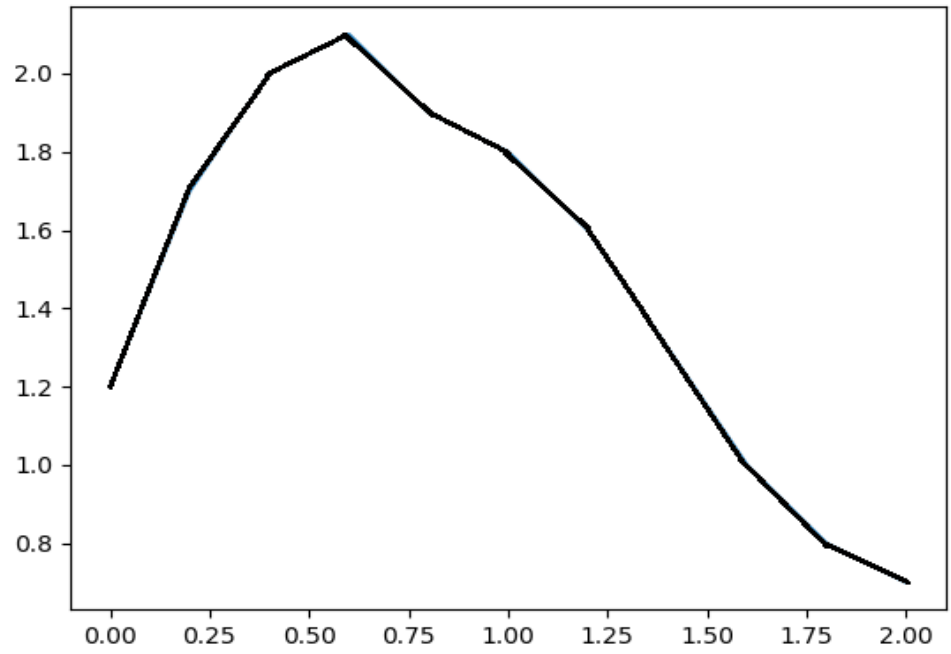

Рис. 4.2

Ординаты точек могут определяться в результате вычислений:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0,2,11)
y = 2 * np.sin(x) + 1
plt.plot(x,y,’o-‘)
plt.show()
```

Можно написать свою собственную функцию для вычислений:

```
import matplotlib.pyplot as plt
import numpy as np
import math
def f(x):
   y = []
   for xx in x:
      yy = 2 * math.sin(xx) + 1
      y.append(yy)
   return y
x = np.linspace(0,2,11)
y = f(x)
plt.plot(x,y)
plt.show()
```

Выведенный рисунок можно сохранить или распечатать.

На одном рисунке можно расположить несколько графиков. Здесь есть два способа. Первый способ заключается в том, чтобы написать несколько вызовов функции ***plot()***:

```
. . .
x = [. . .]
y1 = [. . .]
y2 = [. . .]
plt.plot(x,y1)
plt.plot(x,y2)
. . .
```

Второй способ заключается в том, чтобы использовать следующую форму функции ***plot()***:

*plot(x1,y1[,fmt1], x2,y2[,fmt2],. . . **kwargs)*

Параметры *****kwargs*** действуют на все графики, параметр ***fmt*** относится только к своему графику.

Построим на одном рисунке графики двух функций. С помощью параметра ***color*** изменим цвет первой кривой, а с помощью параметра ***ls*** (сокращенно от ***line style***) зададим стиль линии на «точечный». С помощью параметра ***label*** зададим названия для кривых в легенде. Чтобы легенда появилась, нужно набрать команду ***plt.legend()***. Если у вас установлен ***latex***, то можете в ***label*** указывать строки с математическими формулами, набранными на ***latex***. С помощью команды ***plt.xticks*** заменим числа, выводящиеся по оси ***X***.

```
import numpy as np
import matplotlib.pyplot as plt
def func(x):
    return np.exp(-x**2)
a = -1
b = 1
x_plot = np.linspace(a, b, 100)
plt.plot(x_plot, func(x_plot), color='black', label='f(x)')
plt.plot(x_plot, func(x_plot*2), ls='dotted', label='f(2x)')
plt.legend()
plt.show()
```

В итоге получим изображение на рис. 4.3:

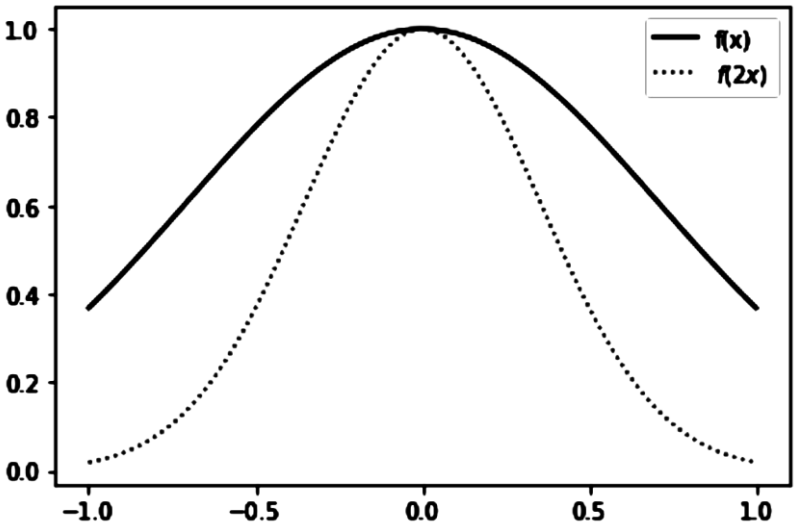

Рис. 4.3

Теперь создадим на одном рисунке две системы координат, расположенные в одном горизонтальном ряду. Это делается с помощью команд ***plt.subplots***. Первыми двумя целочисленными параметрами задается конфигурация графиков (аналогично размеру матрицы). Параметр ***figsize*** отвечает за размер итогового изображения в дюймах. Команда ***plt.subplots*** возвращает два параметра – итоговое изображение и массив подграфиков. В нашем случае он одномерный. На первом подграфике (***axs[0]***) то же самое, что и раньше. На втором подграфике (***axs[1]***) сделаем «увеличение» на области ***x*** в диапазоне [-0.1, 0.1]. Для этого с помощью методов ***set_xlim*** и ***set_ylim*** зададим, в каких пределах изменяются ***x*** и ***y*** соответственно. Обратите внимание, что теперь некоторые команды вызываются по-другому. Результат представлен на рис. 4.4.

```
import numpy as np
import matplotlib.pyplot as plt
def func(x):  return np.exp(-x**2)
a = -1; b = 1
x_plot = np.linspace(a, b, 100)
fig, axs = plt.subplots(1, 2, figsize=(12, 5))
axs[0].plot(x_plot, func(x_plot), color='black', label='f(x)')
axs[0].plot(x_plot, func(x_plot*2), ls='dotted', label='$f(2x)$')
axs[0].set_xticks(np.arange(-1, 1.5, step=0.5))
axs[0].legend()
axs[1].plot(x_plot, func(x_plot), color='black', label='f(x)')
axs[1].plot(x_plot, func(x_plot*2), ls='dotted', label='$f(2x)$')
axs[1].set_xlim((-0.1, 0.1))
axs[1].set_ylim((0.95, 1.01))
axs[1].set_xticks(np.linspace(-0.1, 0.1, 5))
axs[1].legend()
```

```
plt.show()
```

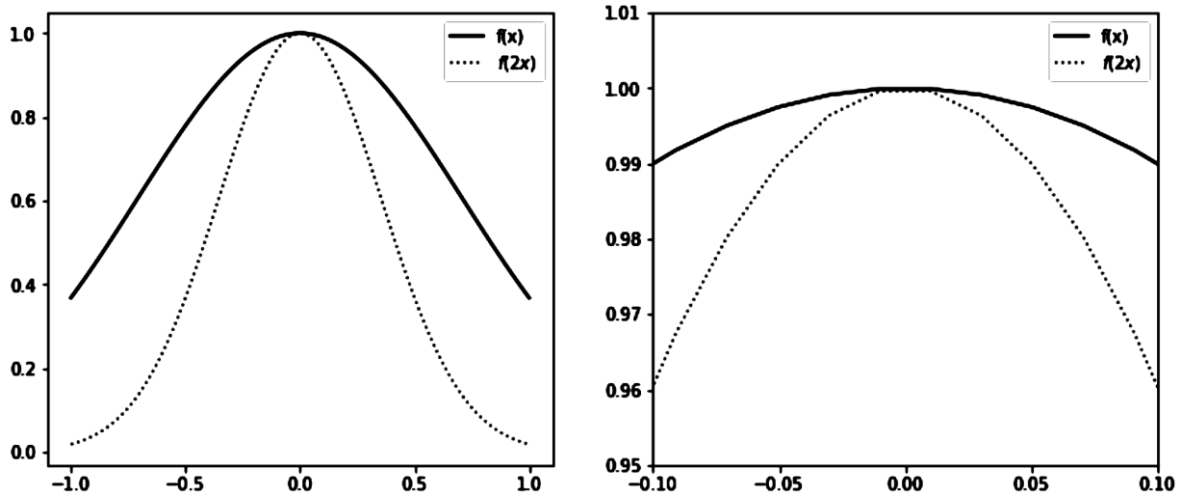

Рис. 4.4

Для построения точечного графика используется параметр ***marker***, указывающий, какой именно маркер использовать. Размер маркера задается параметром ***markersize***. Параметр ***ls*** надо при этом указать пустым. Так же можно менять толщину линий с помощью параметра ***linewidth***.

```
import numpy as np
import matplotlib.pyplot as plt
x_plot = np.linspace(-1, 1, 100)
x_dot = np.array([-0.2, 0.3, 0.7])
y_dot = np.exp(x_dot) + (np.random.rand(3) - 0.5)/2
plt.plot(x_plot, np.exp(x_plot), linewidth=3)
plt.plot(x_dot, y_dot, color='black', ls='', marker='.', mark-
ersize=12)
plt.show()
```

В итоге получим изображение на рис. 4.5:

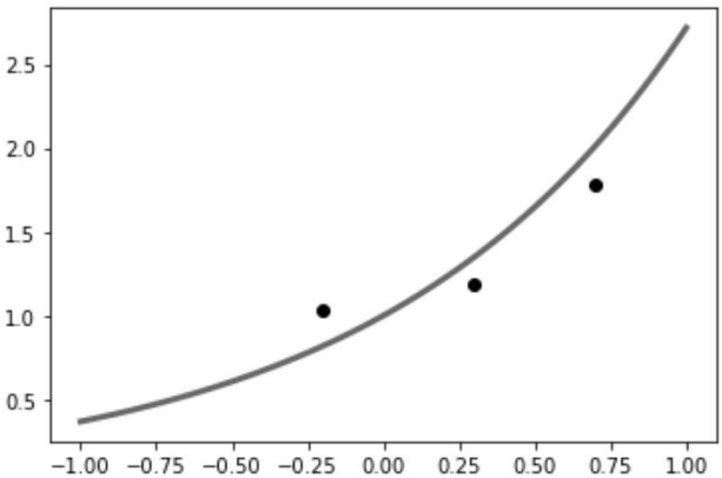

Рис. 4.5

Глава 5. РЕШЕНИЕ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

5.1. Локализация корней

Предположим, уравнение $f(x)=0$ имеет вещественный корень $\bar{x}$. При решении уравнения численно требуется найти такое приближение к корню x_n, что $\left|x_n-\bar{x}\right|\le\varepsilon$. Тогда будем называть x_n значением корня, найденным с заданной точностью ε. Все методы вычисления корня уравнения требуют локализации корня. Это можно сделать графически или таблично. Геометрически корень соответствует точке пересечения графика функции $y=f(x)$ с осью OX. При табличном способе следует найти такой отрезок [a,b], где функция меняет знак ($f(a)f(b)<0$).

Пример 5.1. Рассмотрим уравнение $x^2-arctgx=0$. Построим график и таблицу значений.

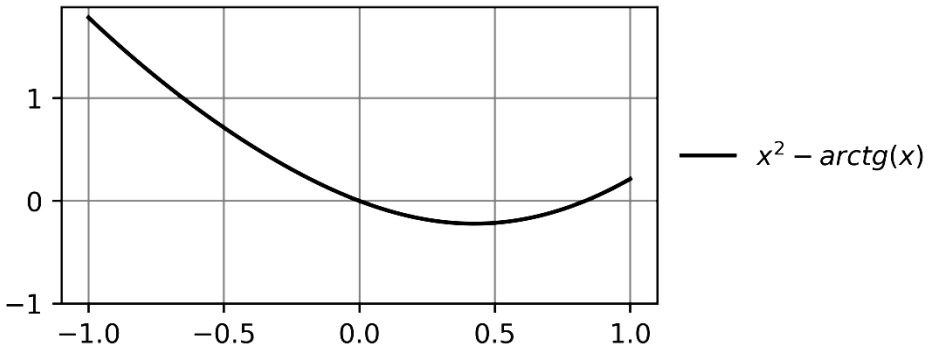

Рис. 5.1

```
def f(x):
    return x**2 - np.arctan(x)
np.set_printoptions(precision=2, floatmode='fixed')
x = np.linspace(-0.5, 1, 9)
print(x, '\n', f(x))
[-0.50 -0.31 -0.12  0.06  0.25  0.44  0.62  0.81  1.00]
[ 0.71  0.40  0.14 -0.06 -0.18 -0.22 -0.17 -0.02  0.21]
```

Видно, что уравнение имеет два корня: $x=0$ и $x\in[0.5,1]$.

Самый простой метод – метод половинного деления – можно рассматривать как улучшение процедуры нахождения корня табличным способом. Деля отрезок пополам и каждый раз выбирая тот из подотрезков, в концах которого функция имеет разные знаки, можно получить решение с заданной точностью. Действительно, если на каком-

то шаге получена длина отрезка локализации $b-a \le 2\varepsilon$, то средняя точка отрезка $c=(a+b)/2$ является приближением к корню с точностью ε, так как независимо от положения точки $\bar{x}$ на последнем отрезке $|c-\bar{x}| \le \varepsilon$.

Для примера **5.1** легко убедиться, что последовательность вложенных отрезков [0.5,1], [0.75,1] , [0.75,0.875] содержит корень уравнения. При выборе

$$x = c = \frac{0.75+0.875}{2} = 0.8125$$

можно считать, что корень найден с точностью 0.0625.

5.2. Метод Ньютона

Рассмотрим один из наиболее популярных методов – метод Ньютона. Он относится к итерационным методам решения, то есть методам, которые порождают последовательность приближений x_n, сходящуюся к корню в смысле $\lim\limits_{n\to\infty}|x_n-\bar{x}|=0$. Величину $e_n=|x_n-\bar{x}|$ называют абсолютной ошибкой на n-ой итерации или погрешностью n-ой итерации. Итерационный метод имеет порядок p, если $e_{n+1} \le Ce_n^p$. Теоретически можно доказать, что метод Ньютона имеет второй порядок сходимости (сходится квадратично). Расчетная формула метода Ньютона имеет вид

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

Использование итерационных методов требует наличия критерия окончания итераций. Если метод сходится сверхлинейно ($p>1$), то критерий окончания имеет вид $|x_{n+1}-x_n| \le \varepsilon$. Заметим, что для начала расчетов по методу Ньютона требуется задать только одно начальное приближение x_0. При вычислении корня с заданной точностью обычно не используют массив приближений. а ограничиваются двумя соседними приближениями x_n и x_{n+1}. В рассматриваемом ниже примере эти переменные обозначены как x0 и x1.

Пример 5.2. Запишем реализацию метода Ньютона с подсчетом числа итераций.

```
def newton(f, df, x0, eps):
    n = 1
    x1 = x0 - f(x0) / df(x0)
    while abs(x1 - x0) > eps:
        x0 = x1
```

```
        x1 = x0 - f(x0) / df(x0)
        n = n + 1
    return x1, n
```

Для примера решим уравнение $x - e^{-x}\cos(x) = 0.$ с точностью $\varepsilon = 10^{-10}$. Тогда $f(x) = x - e^{-x}\cos(x)$ и $f'(x) = 1 + e^{-x}\left(\sin(x) + \cos(x)\right).$

```
def f(x):
    return x - np.exp(-x) * np.cos(x)
def df(x):
    return 1 + np.exp(-x) * (np.sin(x) + np.cos(x))
```

Построим график заданной функции:

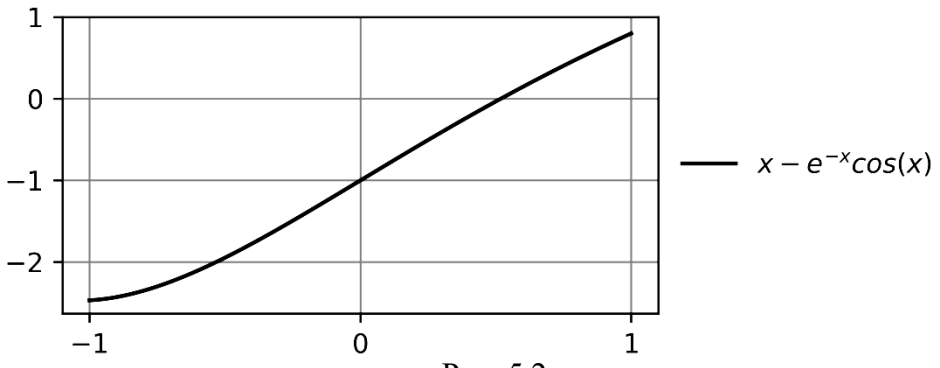

Рис. 5.2

Вызывая

```
newton(f, df, 1, 10e-10)
```

получаем

```
(0.5177573636824584, 5)
```

Модифицировав функцию ***newton*** (аналогично примеру 5.3), выпишем результат всех 5 итераций в виде таблицы (таб. 5.1).

n	x_n	$\lvert x_n - x_{n-1} \rvert$	$\lvert f(x_n) \rvert$
0	1	–	0.8
1	0.46879262368	$5.3 \cdot 10^{-1}$	$8.9 \cdot 10^{-2}$
2	0.51738359537	$4.8 \cdot 10^{-2}$	$6.8 \cdot 10^{-4}$
3	0.51775734096	$3.7 \cdot 10^{-4}$	$4.1 \cdot 10^{-8}$
4	0.51775736368	$2.3 \cdot 10^{-8}$	$1.1 \cdot 10^{-16}$
5	0.51775736368	$1.1 \cdot 10^{-16}$	$2.2 \cdot 10^{-16}$

Таб. 5.1

5.3. Метод простой итерации

Другим популярным методом решения является метод простой итерации. Исходное уравнение $f(x)=0$ заменяют эквивалентным ему уравнением $x=\varphi(x)$. Затем, выбрав некоторое начальное приближение x_0 к корню $\bar{x}$, выполняют итерации по формуле $x_{n+1}=\varphi(x_n)$, n=0,1,2,…. При выполнении условия сходимости метода простой итерации $\max\limits_{[a,b]}|\varphi'(x)|\le q<1$ можно получить решение с заданной точностью ε, если использовать критерий окончания итераций:

$$|x_{n+1}-x_n|\le\frac{1-q}{q}\varepsilon.$$

Однако полезно контролировать процесс вычислений, поэтому можно организовать цикл по числу итераций.

Пример 5.3. При различном выборе преобразования уравнения $x^2-arctgx=0$ к виду, удобному для итераций, метод сходится к разным корням.

```
def g1(x):
    return np.sqrt(np.arctan(x))
def g2(x):
    return np.tan(x ** 2)
def MPI(x0, phi, N):
    res = np.zeros(N + 1)
    res[0] = x0
    for i in range(N):
        res[i + 1] = phi(res[i])
    return res

N = 5
x0 = 0.8
x1 = MPI(x0, g1, N)
x2 = MPI(x0, g2, N)
print(x1)
print(x2)
```

```
[0.8        0.82142616 0.8292586  0.83206291 0.83305945 0.83341263]
[0.8        0.74454382 0.61910019 0.40322685 0.16403998 0.02691561]
```

5.4. Обзор библиотечных функций

В библиотеке ***scipy.optimize*** есть набор функций для решения нелинейных уравнений. Функция ***scipy.optimize.newton*** реализует метод секущих, если методу не передать производную, и метод Ньютона, если передана производная ***fprime***. Если заданы первая производная ***fprime*** и вторая производная ***fprime2***, то реализуется метод Галлея с кубической скоростью сходимости.

Пример 5.4. Решим уравнение $x^2 - arctgx = 0$ с точностью $\varepsilon = 10^{-10}$.

```
from scipy.optimize import newton
x0 = 0.8
def f(x):
    return x ** 2 - np.arctan(x)
def df(x):
    return 2 * x - 1 / (x ** 2 + 1)
def d2f(x):
    return 2 * x / (x ** 2 + 1) ** 2 + 2

res1 = newton(f, x0, tol=10e-10, full_output=True)
res2 = newton(f, x0, tol=10e-10, full_output=True, fprime=df)
res3 = newton(f, x0, tol=10e-10, full_output=True, fprime=df,
fprime2=d2f)
```

Результат работы трех методов представлен ниже.
Метод секущих

```
 (0.8336061944066759,       converged: True
           flag: 'converged'
 function_calls: 6,          iterations: 5
           root: 0.8336061944066759)
```

Метод Ньютона

```
 (0.833606194406676,       converged: True
           flag: 'converged'
 function_calls: 8,        iterations: 4
           root: 0.833606194406676)
```

Метод Галлея

```
 (0.833606194406676,       converged: True
           flag: 'converged'
 function_calls: 9,        iterations: 3
           root: 0.833606194406676)
```

Так же в библиотеке имеются метод бисекции ***scipy.optimize.bisect*** и гибридные методы Брента и Риддера. Функция ***scipy.optimize.root_scalar*** позволяет вызывать каждый из указанных методов, передавая название метода в качестве параметра [5].

Глава 6. РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

6.1. Прямые методы

Рассмотрим систему линейных алгебраических уравнений (СЛАУ)

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n = b_{1,} \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n = b_2, \\ \ldots \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n = b_n \end{cases}$$

с невырожденной матрицей $A = \{a_{i,j}\}$ $(i, j = 1,\ldots,n)$. При выполнении данного условия для любого вектора правых частей b существует единственное решение системы – вектор x.

Один из универсальных методов решения СЛАУ, метод Гаусса, заключается в последовательном исключении неизвестных из нижележащих уравнений. На первом шаге из уравнений со 2-го по *n*-ое исключается неизвестная x_1, затем на втором шаге из уравнений с 3-го по *n*-ое исключается неизвестная x_2 и так далее. Если все главные миноры матрицы системы A отличны от нуля, описанный процесс может быть доведен до конца. В итоге исходная система $Ax = b$ будет приведена к виду $Ux = \tilde{b}$, где матрица U будет иметь верхнетреугольный вид.

Кратко формулы *j*-го шага можно описать следующим образом. Сначала вычисляется коэффициент $\mu_{i,j} = a_{i,j} / a_{j,j}$, затем из *i*-ой строки вычитается *j*-ая строка, умноженная на коэффициент $\mu_{i,j}$. Эти действия выполняются для строк с (*j*+1)-ой до *n*-ой. Под строкой здесь следует понимать строку системы, включающую также элемент правой части. Приведение матрицы системы к треугольному виду называют прямым ходом метода Гаусса. После его выполнения проводится обратный ход, в процессе которого последовательно находятся неизвестные с x_n по x_1. В общем виде этот процесс можно описать формулой $x_n = \tilde{b}_n / a_{n,n}$, $x_i = \left(\tilde{b}_i - a_{i,i+1}x_{i+1} - \ldots - a_{i,n}x_n\right) / a_{i,i}$ $(i = n-1,\ldots,1)$.

Описанный выше алгоритм принято называть схемой единственного деления. В случае, когда один из главных миноров

матрицы системы равен нулю, эта схема порождает нулевой элемент на главной диагонали матрицы A и дальнейший расчет становится невозможен. Для преодоления такой ситуации существуют модификации метода с выбором главного элемента. Подробнее о них можно справиться в [1, 2].

Пример 6.1. Запишем реализацию метода Гаусса, используя средства языка python для обработки массивов.

```
import numpy as np
A = np.array([[3, 2, -2, 3],
              [6, 2, -2, 9],
              [-3, -6, 5, 2],
              [-9, -2, -1, -9]])  # матрица системы
b = np.array([-4, -7, 6, 1])  # вектор правых частей
n = 4  # порядок системы

def gauss(A, b):
    # Прямой ход
    for j in range(n - 1):
        for i in range(j + 1, n):
            m = A[i, j] / A[j, j]
            A[i,j::] = A[i,j::] - m * A[j,j::] # изменяем
                                                # строку матрицы
            b[i] = b[i] - m * b[j] # изменяем правую часть
    # Обратный ход
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = b[i] - np.sum(A[i, i + 1::] * x[i + 1::])
        x[i] /= A[i, i]
    return x

x = gauss(A, b)
print(x)
```

В результате будет выдан ответ [1, -1, 1, -1]. Как несложно проверить, вектор является точным решением системы с прописанными в коде примера матрицей и правой частью.

В библиотеке ***scipy.linalg*** есть набор функций для решения СЛАУ общего и специального вида. В частности, функция ***scipy.linalg.solve()*** реализует метод Гаусса с выбором по столбцу.

Пример 6.2. Решим ту же систему библиотечными средствами.

```
import numpy as np
from scipy import linalg
A = np.array([[3, 2, -2, 3],
```

```
              [6, 2, -2, 9],
              [-3, -6, 5, 2],
              [-9, -2, -1, -9]])
b = np.array([-4, -7, 6, 1])
x = linalg.solve(A, b)
print(x)
```

В результате будет выдан тот же ответ [1, -1, 1, -1].

Некоторые функции библиотеки ***scipy.linalg*** предназначены для решения систем специального вида. Функция ***scipy.linalg.solve_triangular()*** может быть использована для решения систем с треугольной матрицей.

Пример 6.3. Изменим реализацию обратного хода метода Гаусса из примера 6.1 на вызов библиотечной функции.

```
import numpy as np
from scipy import linalg
A = np.array([[3, 2, -2, 3],
              [6, 2, -2, 9],
              [-3, -6, 5, 2],
              [-9, -2, -1, -9]])  # матрица системы
b = np.array([-4, -7, 6, 1])  # вектор правых частей
n = 4  # порядок системы

def gauss(A, b):
    # Прямой ход
    for j in range(n - 1):
        for i in range(j + 1, n):
            m = A[i, j] / A[j, j]
            A[i,j::] = A[i,j::] - m * A[j,j::]
            b[i] = b[i] - m * b[j]
    # Обратный ход
    x = linalg.solve_triangular(A,b,lower=False)
    return x

x = gauss(A, b)
print(x)
```

В результате будет выдан все тот же ответ [1, -1, 1, -1]. Здесь в последнем аргументе функции ***solve_triangular()*** явно указано, какой треугольник матрицы заполнен. При значении ***lower*** = ***True*** для решения будут использоваться только элементы нижнетреугольной части.

Еще одним широко распространенным типом матриц являются ленточные, в которых все ненулевые элементы сосредоточены на нескольких диагоналях, примыкающих к главной. Для их решения можно использовать функцию ***scipy.linalg.solve_banded()***.

Пример 6.4. Решим трехдиагональную систему

$$A=\begin{pmatrix} 3 & 2 & 0 & 0 & 0 \\ 3 & 6 & -2 & 0 & 0 \\ 0 & -3 & -6 & 1 & 0 \\ 0 & 0 & -1 & 5 & 3 \\ 0 & 0 & 0 & -2 & 5 \end{pmatrix}, \quad b=\begin{pmatrix} 3 \\ 1 \\ -6 \\ 2 \\ 5 \end{pmatrix}.$$

```
import numpy as np
from scipy import linalg
A3 = np.array([[0, 2, -2, 1, 3],
               [3, 6, -6, 5, 5],
               [3, -3, -1, -2, 0]])
b = np.array([3, 1, -6, 2, 5])
x = linalg.solve_banded((1, 1), A3, b)
print(x)
```

В результате будет выдан ответ [1, 0, 1, 0, 1], правильность которого несложно проверить самостоятельно. Следует обратить внимание, что матрица, передаваемая в функцию, хранит только ненулевые диагонали, записанные по строкам от самой дальней над- до самой дальней под-диагонали. В данном примере в первой строке записана единственная наддиагональ, прижатая вправо (первый ноль соответствует отсутствующему элементу наддиагонали), во второй строке содержится главная диагональ исходной матрицы, а в третьей – единственная поддиагональ, прижатая влево (отсутствующему элементу поддиагонали соответствует последний ноль). Вектор правых частей задается стандартно. Первый аргумент определяет ширину ленты: сначала указывается количество поддиагоналей, затем наддиагоналей. В случае трехдиагональной матрицы, для которой выполнены условия устойчивости метода прогонки, алгоритм функции ***scipy.linalg.solve_banded()*** эквивалентен классическому алгоритму прогонки.

6.2. Итерационные методы

Все описанные выше методы относятся к классу **прямых** методов. Помимо этого, широкое распространение получили методы **итерационные**, порождающие в процессе своей работы последовательность приближений к точному решению системы. Наиболее простым является метод Якоби (простой итерации). Для его применения следует предварительно привести СЛАУ к виду $x = Bx + c$ и задать некоторое начальное приближение $x^{(0)}$. После этого следующее

приближение определяется формулой $x^{(k+1)} = Bx^{(k)} + c$ $(k = 0,1,2,\ldots)$. Если выполнено условие $\|B\| < 1$, то последовательность приближений сходится к точному решению $\bar{x}$ и верна оценка

$$\left\|x^{(k+1)} - \bar{x}\right\| \le \frac{\|B\|}{1-\|B\|}\left\|x^{(k+1)} - x^{(k)}\right\|,$$

откуда следует критерий окончания итераций

$$\left\|x^{(k+1)} - x^{(k)}\right\| \le \frac{1-\|B\|}{\|B\|}\varepsilon,$$

в котором ε – требуемая точность (по норме). При программной реализации обычно не используют массив приближений, но ограничиваются двумя соседними приближениями $x^{(k)}$ и $x^{(k+1)}$. В рассматриваемом ниже примере эти переменные обозначены как ***x0*** и ***x1***.

Пример 6.5. Запишем реализацию метода Якоби.

```
import numpy as np
from scipy import linalg
A = np.array([[10, 1, 1, 1],
              [1, 10, 1, 1],
              [1, 1, 10, 1],
              [1, 1, 1, 10]])  # матрица системы
b = np.array([2, 11, 11, 2])  # вектор правых частей
eps = 1e-8  # требуемая точность

def jacobi(A, b, eps):
    n = b.size
    B = np.ndarray((n, n))
    c = np.ndarray(n)
    # преобразование к виду x = Bx + c
    for i in range(n):
        B[i] = -A[i] / A[i, i]
        B[i, i] = 0
        c[i] = b[i] / A[i, i]
    # подготовка критерия окончания
    nrmB = linalg.norm(B, 1)
    e1 = eps * (1 - nrmB) / nrmB
    # задание начального приближения
    x0 = c
    # основной цикл метода
    x1 = B @ x0 + c
    while linalg.norm(x1 - x0, 1) > e1:
        x0 = x1
        x1 = B @ x0 + c
    return x1
```

```
x = jacobi(A, b, eps)
print(x)
```

В результате выполнения выдается ответ $\left(6\cdot10^{-10}, 1, 1, 6\cdot10^{-10}\right)$, совпадающий с точным решением $\left(0, 1, 1, 0\right)$ в рамках заданной точности. Здесь использована также функция ***scipy.linalg.norm()***, возвращающая норму передаваемой в нее матрицы. Второй аргумент функции конкретизирует вычисляемую норму. В данном случае используется $\|B\|_1$.

Большое количество специальных методов решения систем содержит библиотека ***scipy.sparse.linalg***. В качестве примера рассмотрим применение метода сопряженных градиентов для решения той же системы. Этот метод гарантированно сходится для СЛАУ с симметричными и положительно определенными матрицами. В рассматриваемом примере симметричность очевидна, а положительная определенность следует из диагонального преобладания в матрице A.

Пример 6.6. Решение системы методом сопряженных градиентов.

```
import numpy as np
from scipy.sparse import linalg
A = np.array([[10, 1, 1, 1],
              [1, 10, 1, 1],
              [1, 1, 10, 1],
              [1, 1, 1, 10]])    # матрица системы
b = np.array([2, 11, 11, 2])     # вектор правых частей
x = linalg.cg(A, b, tol = 1e-8)
print(x)
```

6.3. Поиск собственных значений

Другой важной задачей линейной алгебры является поиск **собственных чисел** квадратной матрицы. Как известно, число λ и ненулевой вектор e называются собственным числом и собственным вектором матрицы A, если $Ae=\lambda e$. Из данного определения следует, что $\left|A-\lambda I\right|=0$, где через I обозначена единичная матрица. Для поиска этих величин можно воспользоваться библиотечной функцией ***scipy.linalg.eigvals()***, которая возвращает массив всех собственных чисел матрицы. Количество элементов в массиве определяется порядком матрицы, а его элементы являются комплексными числами.

Пример 6.7. Поиск собственных чисел матрицы.

```
import numpy as np
```

```
from scipy import linalg
A = np.array([[10, 1, 1, 1],
              [1, 10, 1, 1],
              [1, 1, 10, 1],
              [1, 1, 1, 10]])
y = linalg.eigvals(A) #поиск всех собственных чисел
print(y)
print(linalg.det(A-y[1]*np.eye(4)))   #выборочная проверка
```

В результате выполнения этого кода будет выдан массив четырех собственных чисел [9.+0.j 13.+0.j 9.+0.j 9.+0.j]. Выборочная проверка демонстрирует правильность работы программы.

Поскольку симметричность рассматриваемой в примере матрицы гарантирует вещественность всех ее собственных чисел, можно исключить работу с комплексными числами, сразу взяв их действительные части. Для этого к результату вызова функции можно добавить соответствующий метод: ***y = linalg.eigvals(A).real***.

Глава 7. ПРИБЛИЖЕНИЕ ФУНКЦИЙ

7.1. Интерполяция многочленами

В наиболее общем виде задачу приближения функций одного аргумента можно сформулировать следующим образом. По заданным узлам $x_0, x_1, \ldots, x_n$ и заданным в них значениям некоторой функции $y_0, y_1, \ldots, y_n$ требуется построить функцию непрерывно меняющегося аргумента $G(x)$, такую что $G(x_i) \approx y_i$, $i = 0, \ldots, n$. В зависимости от того, в каком смысле понимается приближенное равенство, возникают различные конкретные задачи приближения со своими специфическими методами их решения.

Начнем с задачи **глобальной интерполяции многочленами**. В этом случае искомым является многочлен $P_n(x)$ степени n, который принимает в каждом узле строго заданное значение: $P_n(x_i) = y_i$, $i = 0, \ldots, n$. Если все узлы интерполяции различны, то для любого набора значений функции в них существует единственный интерполяционный многочлен.

Существует несколько способов построения глобального интерполяционного многочлена. Один из них – многочлен Лагранжа, определяемый формулой

$$L_n(x) = \sum_{k=0}^{n} y_k \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}.$$

Пример 7.1. Программная реализация многочлена Лагранжа.

```
import numpy as np

def Lagr(t, x, y):    #вычисление значения многочлена Лагранжа
  n = x.size
  s = 0
  for k in range(n):
    p = 1
    for j in range(n):
      if (j != k): p*=(t - x[j]) / (x[k] - x[j])
    s += y[k] * p
  return s

x = np.linspace(-1, 1, 7)        # массив узлов интерполяции
y = 1. / (1. + 25 * x**2)         # массив значений функции
```

В результате исполнения кода будет получен следующий график.

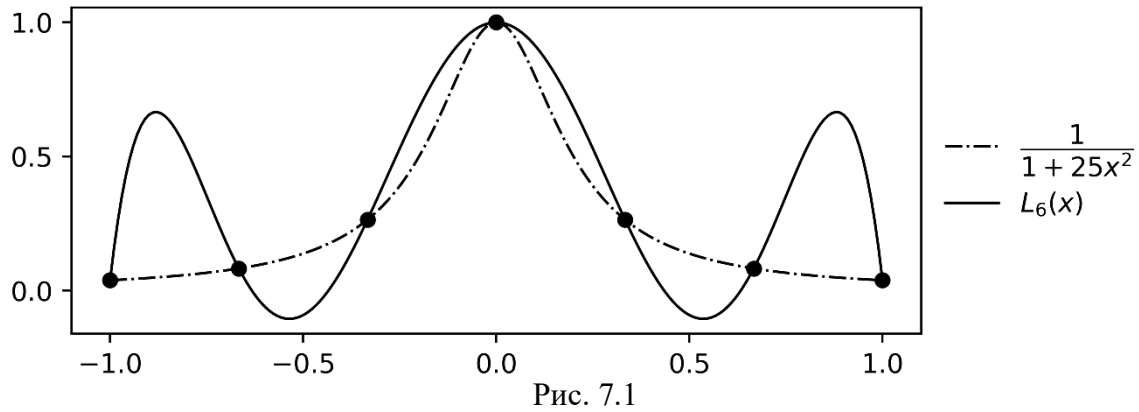

Рис. 7.1

Тот же результат может быть получен средствами математических библиотек, а именно, функциями из модуля ***interpolate*** пакета ***scipy***.

Пример 7.2. Использование интерполяционного многочлена Лагранжа из библиотечных функций.

```
import numpy as np
from scipy import interpolate

x = np.linspace(-1, 1, 7)              # массив узлов интерполяции
y = 1. / (1. + 25 * x**2)              # массив значений функции
```

```
p = interpolate.lagrange(x, y)          # интерполяционный многочлен
```

В результате исполнения кода будет получен график, совпадающий с приведенным выше.

7.2. Интерполяция сплайнами

Иной результат (зачастую, более адекватный) может быть получен при использовании кусочно-полиномиальной интерполяции, например, **интерполяции сплайнами**. При таком подходе ищется функция $S(x)$, которая на каждом отрезке $[x_i, x_{i+1}]$ является многочленом (на разных отрезках – разным) некоторой фиксированной степени. При этом в каждом узле интерполяции должно обеспечиваться гладкое сопряжение двух стыкующихся в нем многочленов.

Для построения интерполяционного сплайна можно использовать функцию ***scipy.interpolate.interpld(x, y, kind)***, первые два аргумента которой задают узлы и значения интерполируемой функции, третий определяет степень сплайна (для нулевой степени – некоторые дополнительные параметры).

Пример 7.3. Использование сплайна 1-й степени.

```
import numpy as np
from scipy import interpolate

x = np.linspace(-1, 1, 7)
y = 1. / (1. + 25 * x**2)

p1 = interpolate.interp1d(x, y, kind=1) # сплайн 1-й степени
```

В результате будет получен график

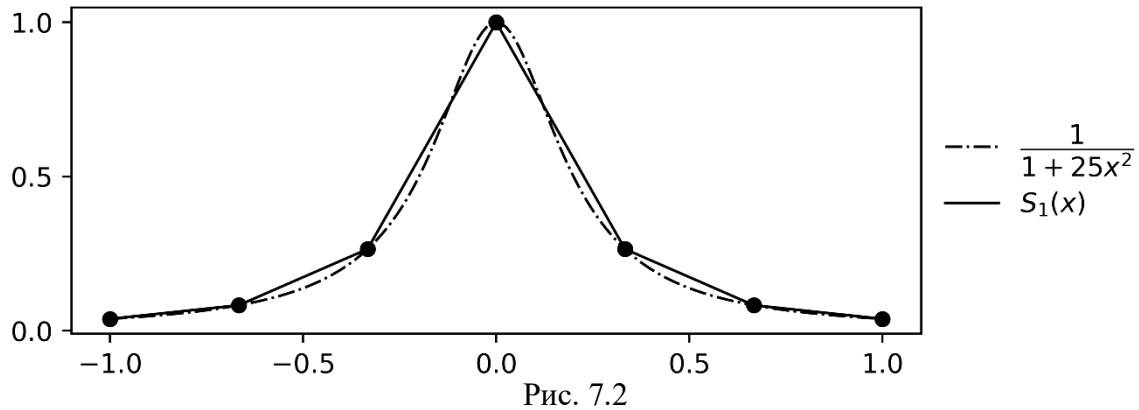

Рис. 7.2

Существует также функция построения кубических сплайнов ***scipy.interpolate.CubicSpline(x, y, bc_type)***. Третий ее аргумент одновременно определяет тип граничных условий и задает дополнительные значения на границе. Например, запись ***bc_type*** = ((2, 0.0), (2, 0.0)) означает, что на левом и на правом концах отрезка интерполяции $[x_0, x_n]$ задаются значения второй производной, равные нулю (так называемый, естественный сплайн). Условия слева и справа можно произвольно сочетать друг с другом, соблюдая синтаксис: первое число пары (целое) задает порядок, второе (вещественное) – значение производной указанного порядка.

Пример 7.4. Использование сплайнов 3-й степени.

```
import numpy as np
from scipy import interpolate

x = np.linspace(-1, 1, 7)              # массив узлов интерполяции
y = 1. / (1. + 25 * x**2)              # массив значений функции
p3 = interpolate.interp1d(x, y, kind=3)  # сплайн 3-й степени
pN = interpolate.CubicSpline(x, y, bc_type=((2, 0.0), (2, 0.0)))
```

В результате будет получен график

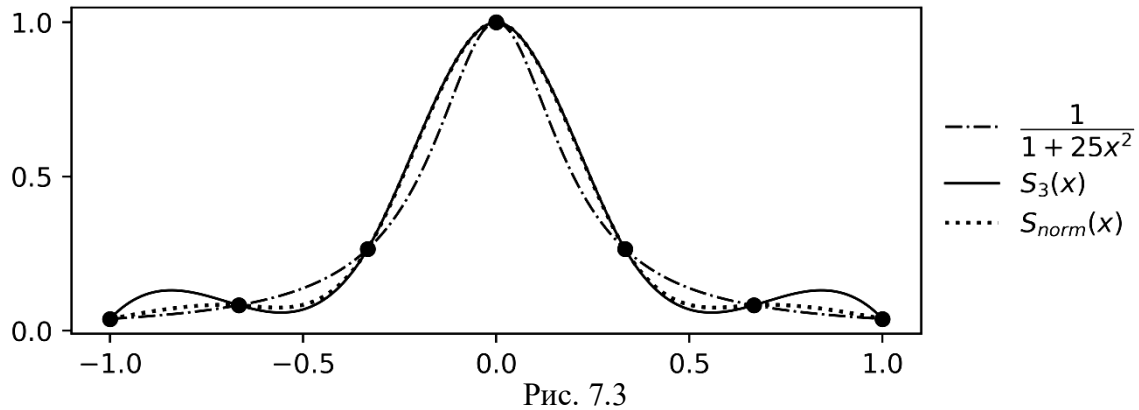

Рис. 7.3

Видно, что различный тип граничных условий существенно меняет вид сплайна на краях области его построения.

7.3. Аппроксимация методом наименьших квадратов

Другим подходом к приближению функций является постановка задачи в смысле **наименьших квадратов**. В этом случае не требуется точного совпадения искомой функции $G(x)$ со значениями функции в узлах интерполяции. Вместо этого ищется функция (заданного вида), которая минимизирует величину

$$\sigma = \sqrt{\frac{1}{n+1}\sum_{i=0}^{n}\left(G(x_i) - y_i\right)^2},$$

называемую среднеквадратичным отклонением.

Как правило, аппроксимирующая функция ищется в виде

$$G(x) = a_0\varphi_0(x) + a_1\varphi_1(x) + \ldots + a_m\varphi_m(x),$$

где $\varphi_k(x)$ $(k = 0,\ldots,m)$ – заданные базисные функции (например, при выборе $\varphi_k(x) = x^k$ имеем классический многочлен).

Если составить матрицу значений базисных функций в узлах аппроксимации $A_{i,k} = \varphi_k(x_i)$ $(i = 0,\ldots,n;\ k = 0,\ldots,m)$, то сформулированная выше задача приближения совпадет с задачей решения переопределенной системы $Aa = y$ в смысле наименьших квадратов (как правило, количество базисных функций существенно меньше количества узлов аппроксимации). Искомым здесь является вектор коэффициентов $a = (a_0,\ a_1,\ \ldots,\ a_m)^T$.

Для решения такой системы можно, предварительно сформировав матрицу, перейти к симметризованной системе $A^T Aa = A^T y$ и решить ее одним из стандартных методов решения СЛАУ. Альтернативным способом является использование функции ***scipy.linalg.lstsq()***, которая решает непосредственно переопределенную систему. Эта функция возвращает кортеж из четырех значений: массив искомых коэффициентов и полученный минимум квадрата невязки $\|Aa - y\|_2^2$, а также информацию о ранге и сингулярных числах матрицы [3].

Пример 7.5. Аппроксимация в смысле наименьших квадратов многочленом первой степени (линейная регрессия).

```
import numpy as np
from scipy import linalg

def G(t, a):
    m = a.size
    s = a[-1]
    for k in range(m - 2, -1, -1):   s = s * t + a[k]
    return s    #значение многочлена с коэффициентами a[]
n = 7           #количество точек
x = np.linspace(0, 1, n)
y = 1. / (1. + 25 * x**2)
m = 1           #степень аппроксимирующего многочлена
A = np.zeros((n, m + 1))
```

```
for i in range(n):
    for k in range(m + 1):
        A[i, k] = x[i]**k
a, res, rnk, sgv = linalg.lstsq(A, y)
print('res  ', res)
```

В результате будет выдано значение квадрата невязки 0.17 и график

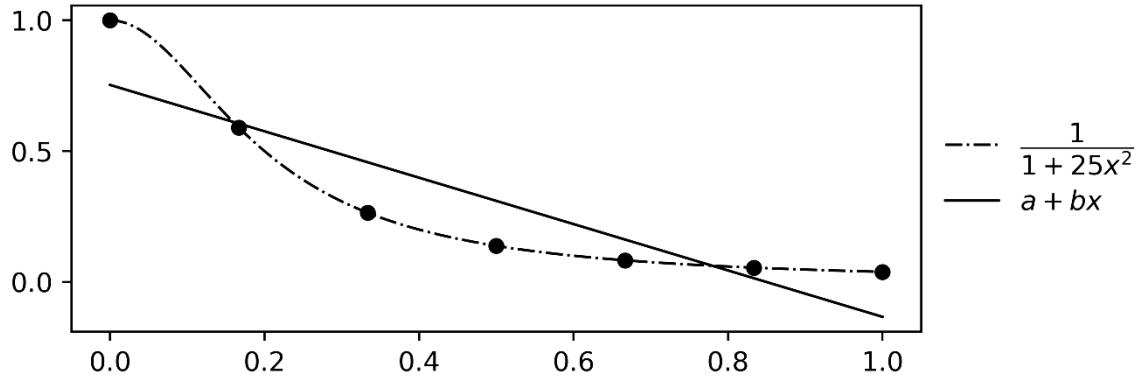

Рис. 7.4

Глава 8. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ И ДИФФЕРЕНЦИРОВАНИЕ

8.1. Формулы интерполяционного типа. Оценка погрешности.

Пусть требуется найти определенный интеграл $\int_a^b f(x)dx$ с точностью ε. Для приближенного значения интеграла чаще всего подынтегральную функцию заменяют близкой вспомогательной функцией, интеграл от которой вычисляется аналитически. Если функция $f(x)$ интерполируется многочленом выбранной степени m на элементарном отрезке $[x_{i-1}, x_i]$, то справедливо соотношение

$$I = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} \left(P_{m,i}(x) + R_{m,i}(x)\right)dx = I_N^h + R_N^h$$

За приближенное значение интеграла принимают значение от вспомогательной функции I_N^h, а величина R_N^h называется остаточным членом квадратурной формулы. Например, при $m = 2$ получаем формулу трапеций с остаточным членом второго порядка точности по h.

$$I_N^h = h\left(\frac{f_0 + f_N}{2} + \sum_{i=1}^{N-1} f_i\right), \qquad R_N^h = \frac{M_2(b-a)}{12}h^2.$$

Для оценки погрешности используется правило Рунге – правило двойного пересчета: интеграл вычисляется с шагом h, затем с

половинным шагом $h/2$. Тогда за погрешность значения $I_{2N}^{h/2}$ принимают величину $I - I_{2N}^{h/2} \approx \frac{I_{2N}^{h/2} - I_N^h}{2^p - 1}$, где p – порядок точности метода.

Пример 8.1. Рассмотрим применение формулы трапеций для вычисления интеграла $I = \int_0^5 e^x \sin x dx$. Его можно вычислить аналитически: $I = -91.708091$ (с учетом 6 знаков после запятой).

Составим программу, вычисляющую приближенное значение интеграла в зависимости от N – числа разбиений отрезка.

```
import numpy as np

def f(x):
    return np.exp(x) * np.sin(x)

def strapz(f, a, b, N):
    h = (b - a) / N
    res = 0.5 * (f(a) + f(b))
    for i in range(1, N):
        res += f(a + i * h)
    res *= h
    return res
```

Вычислим значения интеграла и погрешности при *N=10* и при *N=20*.

```
I = -91.708091
S1 = strapz(f, 0, 5, 10)
S2 = strapz(f, 0, 5, 20)
print(S1, S2)
print(abs(S1 - I), abs(S2 - I))

-93.848423 -92.237253
2.140332 0.529162
```

Найдем также оценку погрешности по Рунге, выполним уточнение по Рунге и найдем погрешность уточненного значения:

```
RR = (S2 - S1) / 3
S_u = S2 + RR
print(S_u, abs(S_u - I))

-91.70019714855765 0.007893851442347
```

Дополним результаты априорной оценкой погрешности. Для этого сначала построим график модуля второй производной подынтегральной функции (рис 8.1).

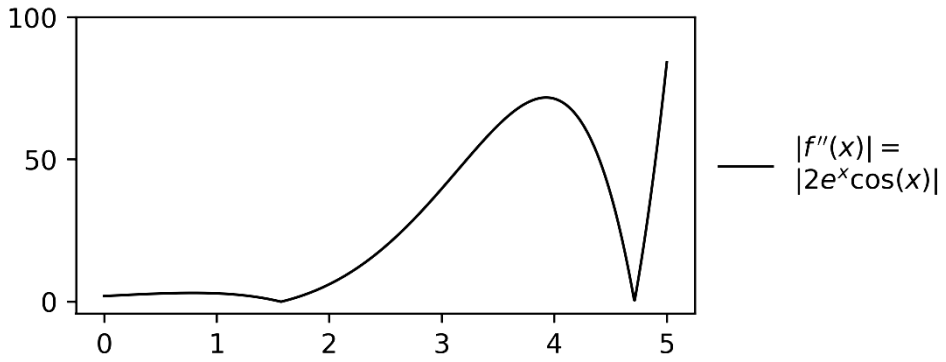

Рис. 8.1

Из графика модуля второй производной видно, что ее максимум $M_2 \approx 84.2$ достигается при $x = 5$. Тогда априорная оценка погрешности

$$R_N = \frac{M_2(b-a)}{12}\left(\frac{b-a}{N}\right)^2$$

дает значения $R_{10} = 8.770667$, $R_{20} = 2.192667$.

Анализируя полученные результаты, можно увидеть что:

1) априорная оценка существенно завышена;

2) правило Рунге дало правильный порядок погрешности;

3) при выполнении уточнения по Рунге погрешность уменьшилась на 2 порядка.

8.2. Библиотечные функции

Функция ***scipy.integrate.quad*** позволяет вычислить значение однократного интеграла таким образом, чтобы выполнялась оценка $\left|I - I_{quad}\right| \le \max\left(\textbf{\textit{epsabs, epsrel}} \cdot \left|I\right|\right)$, где параметры ***epsabs*** и ***epsrel*** соответствуют абсолютной и относительной погрешности. По умолчанию ***epsabs*** и ***epsrel*** заданы как $1.49 \cdot 10^{-8}$. В качестве результата функция возвращает значение интеграла и величину погрешности.

Пример 8.2. Вычисление интеграла из примера 8.1 с помощью функции ***quad***.

```
import numpy as np
from scipy.integrate import quad
def f(x):
     return np.exp(x) * np.sin(x)
```

```
res = quad(f, 0, 5)
print(res)
```

```
(-91.70809100272082, 1.2867755076681554e-12)
```

Функция ***quad*** также позволяет вычислять интегралы с бесконечными пределами интегрирования.

Пример 8.3. Вычислим интеграл $\int_0^{\infty} e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2} \approx 0.886227$.

```
import numpy as np
from scipy.integrate import quad
def f(x):
    return np.exp(-x**2)
res = quad(f, 0, np.inf)
print(res)
```

```
(0.8862269254527579, 7.101318390472462e-09)
```

Для кратного интегрирования есть функции ***scipy.integrate.dblquad, scipy.integrate.tplquad*** и ***scipy.integrate.nquad.***

Рассмотрим ***scipy.integrate.dblquad***. Параметры ***a***, ***b***, ***gfun*** и ***hfun*** соответствуют пределам интегрирования

$$\int_{a}^{b}\int_{gfun(x)}^{hfun(x)} f(x,y)\,dydx.$$

Пример 8.4. Покажем, как можно вычислить повторный интеграл

$$\int_{-1}^{1}\int_{0}^{\sqrt{1-x^2}} \sqrt{1-x^2-y^2}\,dydx$$

с помощью функции ***scipy.integrate.dblquad***. Интеграл представляет объем четверти шара и его точное значение равно $\pi/3 \approx 1.047198$.

```
import numpy as np
from scipy.integrate import dblquad
def f(y, x):
    return np.sqrt(1 - x ** 2 - y ** 2)
def g(x):
    return 0
def h(x):
    return np.sqrt(1 - x ** 2)
res = dblquad(f, -1, 1, g, h)
print(res)
```

```
(1.0471975511965979, 8.833911380179416e-11)
```

Как видим, значения практически совпадают.

8.3. Численное дифференцирование

Для численного дифференцирования в библиотеке ***scipy*** есть функция ***scipy.misc.derivative***. Она реализует дифференцирование по формуле

$$f^{(n)}(x) = \sum_{k=-N}^{N} a_k f(x + kh)$$

где коэффициенты a_k подбираются для достижения максимального порядка точности при заданном количестве узлов.

Параметр ***dx*** соответствует шагу h , ***n*** – порядок производной, ***order*** – количество узлов (только нечетное; не меньше 3).

Пример 8.5. Вычислим вторую производную $f(x) = e^x \sin x$ с разными порядками точности в точке $x = 2$ с шагом $h = 0.1$. Точное значение второй производной $f''(2) = 2\,\mathrm{e}^2 \cos 2 = -6.149864641$.

```
import numpy as np
from scipy.misc import derivative

def f(x):
    return np.exp(x) * np.sin(x)

d2 = -6.149864641
res3 = derivative(f, 2, dx=0.1, n=2, order=3)
res5 = derivative(f, 2, dx=0.1, n=2, order=5)
res7 = derivative(f, 2, dx=0.1, n=2, order=7)
print(res3, abs(res3 - d2))
print(res5, abs(res5 - d2))
print(res7, abs(res7 - d2))

-6.172253968421869 0.02238932742186961
-6.149892080613194 2.7439613194069068e-05
-6.149864449585935 1.9141406504274983e-07
```

Как и ожидалось, с ростом числа узлов точность растет.

Глава 9. РЕШЕНИЕ ЗАДАЧИ КОШИ

9.1. Дискретизация задачи Коши

Задача Коши состоит в том, что требуется найти функцию $y(t)$, удовлетворяющую дифференциальному уравнению и начальному условию:

$$\begin{cases} y'(t) = f(t, y(t)), t > t_0 \\ y(t_0) = y_0 \end{cases}$$

Геометрически задача Коши состоит в нахождении интегральной кривой, которая выходит из точки (t_0, y_0) и в каждой точке (t, y) имеет заданное направление касательной $f(t, y(t))$. Для численного решения задачи отрезок $[t_0, T]$ – область непрерывного изменения аргумента t – заменяется множеством дискретных значений: $t_0 < t_1 < ... < t_N = T$, называемых сеткой. Для простоты будем считать, что сетка равномерная, то есть, $h = (T - t_0) / N$, $t_i = t_0 + ih$, i=0,1, …N.

Следующий этап состоит в замене задачи Коши ее дискретным аналогом – уравнением вида

$$\frac{1}{h}\sum_{j=0}^{k} \alpha_j y_{i+1-j} = \Phi(t_i, y_{i+1-k}, ..., y_i, y_{i+1}, h)$$

где $y_0, y_1, y_2, ...y_{k-1}$ известны. Левую часть этого уравнения можно рассматривать как разностную аппроксимацию производной y', а правую часть Ф можно рассматривать как специальным образом построенную аппроксимацию функции f.

Метод сходится с p-ым порядком точности по h, если справедливо неравенство $E(h) = \max_i \left| y(t_i) - y_i \right| \le ch^p$, где с – константа, не зависящая от h.

Дискретная задача называется также разностной схемой.

Разностная схема аппроксимирует дифференциальную задачу с порядком p, если для функции погрешности аппроксимации

$$\psi_i^h = \frac{1}{h}\sum_{j=0}^{k} \alpha_j y(t_{i+1-j}) - \Phi(t_i, y(t_{i+1-k}), ..., y(t_i), y(t_{i+1}), h)$$

справедливо неравенство: $\Psi = \max_i \left| \psi_i^h \right| \le Ch^p$, где С – константа, не зависящая от h.

9.2. Одношаговые методы

Самым простым методом решения задачи Коши является метод первого порядка точности – метод Эйлера, расчетная формула которого имеет вид:

$$y_{i+1} = y_i + hf(t_i, y_i),\ i = 0,1,\ldots,N-1.$$

Этот метод относится к группе одношаговых методов, в которых для расчета приближенного решения в следующей точке (t_{i+1}, y_{i+1}) требуется информация только о последней вычисленной точке (t_i, y_i).

Пример 9.1. Реализация метода Эйлера.

```
def euler(f, y0, t0, h, n):
    y = np.ndarray(n)
    y[0] = y0
    for i in range(n - 1):
        y[i + 1] = y[i] + h * f(t0 + i * h, y[i])
    return y
```

Для нахождения решения конкретной задачи, например, следующей,

$$\begin{cases} y'(t) = y - (t-1)^2 \\ y(0) = 1 \end{cases}$$

с известным аналитическим решением $y(t) = t^2 + 1$ нужно задать входные данные задачи и обратиться к процедуре. Ниже (таб. 9.1) представлен фрагмент решения задачи методом Эйлера в последующих 7 точках отрезка при шаге $h = 0.25$. Для сравнения дано аналитическое решение в тех же точках и величина погрешности на шаге.

```
def f(t, y):
    return y - (t - 1.0) ** 2

res = euler(f, 1, 0, 0.25, 8)
```

t	0	0.25	0.5	0.75	1.0	1.25	1.5	1.75
$y(t)$	1	1.0625	1.250	1.5625	2.0	2.5625	3.250	4.0625
y^h	1	1.0	1.1094	1.3242	1.6396	2.0496	2.5463	3.1204
r^h	0	0.0625	0.1406	0.2383	0.3604	0.5129	0.7037	0.9421

Таб. 9.1

Для практической оценки погрешности расчета можно использовать правило Рунге. Для этого проводят вычисления с шагами h и $h/2$ и сравнивают величины y_i^h и $y_{2i}^{h/2}$ Апостериорная оценка решения, полученного с шагом $h/2$ методом р-го порядка точности, имеет вид:

$$y(t) - y^{h/2} \approx \frac{y^{h/2} - y^h}{2^p - 1}.$$

В пакете ***scipy.integrate*** представлена функция ***solve_ivp***, позволяющая обращаться к некоторым наиболее популярным методам решения задачи Коши. Параметры метода: ***f*** – функция $f(t, y)$; ***t_span*** – левая и правая граница временного интервала, на котором ищется решение; ***y0*** – начальное значение; ***method*** – метод, с помощью которого ищется решение.

Все предлагаемые в библиотеке методы являются адаптивными и подбирают шаг таким образом, чтобы для локальной погрешности выполнялось условие $l_i \le \mathrm{a\_tol} + \mathrm{r\_tol} \cdot |y_i|$, где ***a_tol*** и ***r_tol*** играют роль абсолютной и относительной погрешности. По умолчанию ***a_tol***=10^{-6} и ***r_tol***= 10^{-3}.

Если требуется получить решение в конкретных промежуточных точках, можно передать их в качестве массива, присвоив параметру ***t_eval***. По умолчанию ***method='RK45'***, что соответствует явному методу Рунге-Кутты 4(5).

Существуют также другие методы, в том числе и неявные, подходящие для решения жестких задач. Если передать ***solve_ivp*** параметр ***dense_output=True***, то метод вернет непрерывное решение на заданном временном отрезке, которое представляет собой интерполяцию сплайном (вид сплайна зависит от метода) сеточного решения.

Пример 9.2. Воспользуемся встроенными средствами Python для решения задачи из примера 9.1 методом Рунге-Кутты 4-го порядка точности (в таблице оставлены 3 знака после запятой).

```
from scipy.integrate import solve_ivp

t = np.linspace(0, 2, 9)
res = solve_ivp(f, t_span=[0, 2], y0=[1], t_eval=t)
print(res['t'], res['y'][0])
```

```
[0.000 0.250 0.500 0.750 1.000 1.250 1.500 1.750 2.000]
[1.000 1.063 1.250 1.563 2.000 2.563 3.250 4.063 5.001]
```

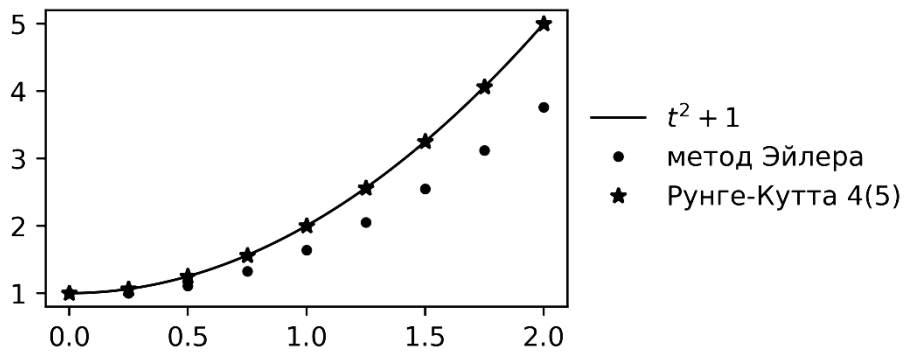

Рис. 9.1

Глобальная погрешность Рунге-Кутта 4(5) составила 5×10^{-3}, а для метода Эйлера 1.24.

Для достижения той же точности методом Эйлера, что в полученном примере методом Рунге-Кутты, потребовалось бы уменьшить шаг в 4096 раз.

Теперь продемонстрируем работу функции ***solve_ivp*** для решения системы дифференциальных уравнений.

Пример 9.3. Решим задачу Коши для системы

$$\begin{cases} y_1' = y_2 \\ y_2' = -y_1 - \cos(t) \\ y_1(0) = 0 \;\; y_2(0) = 0 \end{cases}$$

Входные данные задаем в виде векторов.

```
def f(t, y):
    res = np.ndarray(2)
    res[0] = y[1]
    res[1] = -y[0] - np.cos(t)
    return res

from scipy.integrate import solve_ivp

t = np.linspace(0, 1, 8)
res = solve_ivp(f, t_span=[0, 1], y0=[0, 0], t_eval=t)
print(res['t'], res['y'][0], res['y'][1])
```

t	0	0.125	0.250	0.375	0.500	0.625	0.750	0.875
y_1	0	–0.008	–0.031	–0.069	–0.120	–0.183	–0.256	–0.336
y_2	0	–0.124	–0.245	–0.358	–0.459	–0.546	–0.615	–0.664

Таб. 9.2

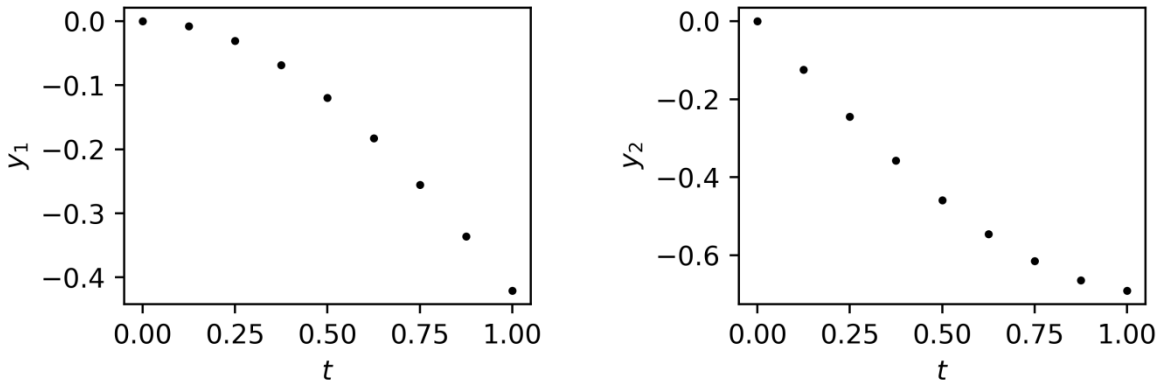

Рис. 9.2

Если найти решение системы на большем отрезке (например, на [0,20], рис. 9.3), то можно дополнительно построить фазовую траекторию системы (рис. 9.4).

```
t = np.linspace(0, 20, 401)
res = solve_ivp(f, t_span=[0, 20], y0=[0, 0], t_eval=t)
```

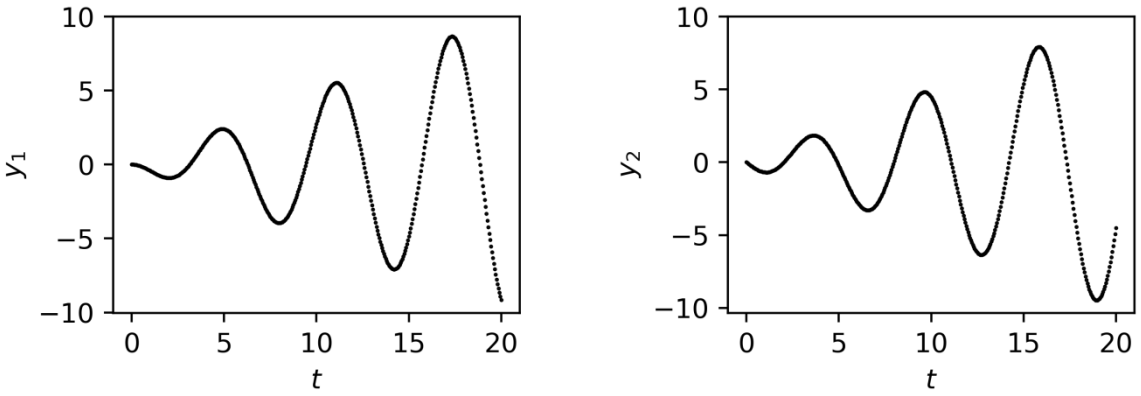

Рис. 9.3

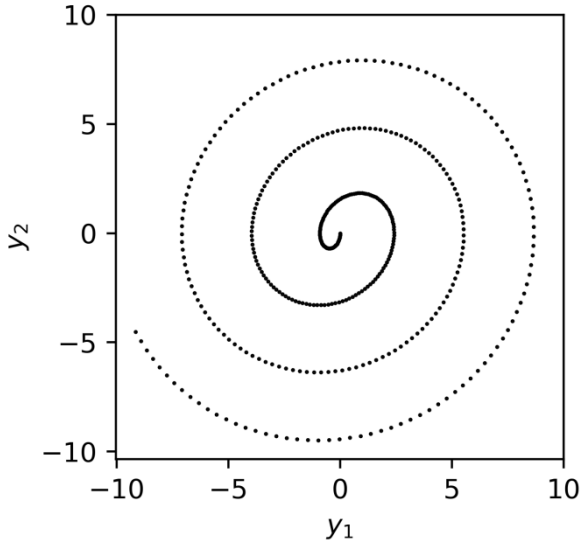

Рис. 9.4

9.3. Многошаговые методы

Рассмотрим метод Адамса-Башфорта четвертого порядка точности:

$$y_{i+1} = y_i + \frac{h}{24}\left(55 f(t_i, y_i) - 59 f(t_{i-1}, y_{i-1}) + 37 f(t_{i-2}, y_{i-2}) - 9 f(t_{i-3}, y_{i-3})\right)$$

Этот метод относится к группе четырехшаговых методов, так как для нахождения решения в точке (t_{i+1}, y_{i+1}) используется информация о четырех предыдущих точках.

Порядок точности данного метода равен 4, так же, как у метода Рунге-Кутты 4-го порядка. В методе Адамса точность достигается за счет использования информации о предыдущих точках, а в методе Рунге-Кутты недостающую информацию получают в результате вычислений в специальным образом выбранных дополнительных точках. Таким образом, многошаговые методы более экономичны, чем одношаговые, однако для начала расчетов потребуется вычислить 4 стартовые точки.

Для того чтобы сохранялся порядок точности выбранного метода p , следует вычислять значения решения в стартовых точках с тем же порядком точности p. Покажем, что несоблюдение этого правила приводит к потере точности.

Пример 9.4. Задача Коши:

$$\begin{cases} y'(t) = y + e^t \cos t \\ y(\pi / 2) = e^{\pi/2} \end{cases}$$

имеет аналитическое решение $y(t) = e^t \sin t$.

Приведем (таб. 9.3) результаты вычислений по методу Адамса-Башфорта 4-го порядка точности в случае нахождения стартовых точек методом Эйлера (yAE) и методом Рунге-Кутты 4-го порядка (yARK). Для сравнения даны значения точного решения задачи.

yt	4.810	5.742	6.582	7.209	7.460	7.133	5.980	3.710	0
yAE	4.810	5.755	6.661	7.433	7.754	7.474	6.394	4.217	0.614
yARK	4.810	5.742	6.582	7.209	7.462	7.137	5.983	3.713	$1.4 \cdot 10^{-4}$

Таб. 9.3

Погрешность в последней точке отрезка $T = \pi$ в первом случае составляет ≈ 0.6, во втором случае ≈ 0.0001.

9.4. Неявные методы

Малые ошибки, появившиеся в начале вычислений, могут совершенно исказить решение, если использовать неподходящий численный метод. Продемонстрируем нарастание погрешностей в следующем примере.

Пример 9.5. Рассмотрим задачу Коши для уравнения 2-го порядка

$$\begin{cases} y''(t) = 25y + 26\cos t \\ y(0) = 0 \quad y'(0) = -5 \end{cases}$$

Легко проверить, что решением задачи является функция $y(t) = e^{-5t} - \cos t$.

Приведем уравнение к системе уравнений первого порядка стандартным преобразованием. Для этого введем функции $y_0 = y$, $y_1 = y'$. Тогда уравнение преобразуется к эквивалентной системе

$$\begin{cases} y_0' = y_1 \\ y_1' = 25y_0 + 26\cos t \\ y_0(0) = 0, \quad y_1(0) = -5 \end{cases}$$

Используем для решения задачи методы Адамса 4-го порядка. Стартовые точки зададим без погрешности, используя значения точного решения.

Ниже приведена программа, реализующая метод Адамса-Моултона с прогнозом по методу Адамса-Башфорта. Если убрать оператор, отвечающий за коррекцию по неявному методу, то получим программу, реализующую явный метод. Эта же программа (без коррекции) использована в примере 9.4.

```
def f(t, y):
    res = np.ndarray(2)
    res[0] = y[1]
    res[1] = 25 * y[0] + 26 * np.cos(t)
    return res
def y(t):
    return np.exp(-5 * t) - np.cos(t)
def dy(t):
    return -5 * np.exp(-5 * t) + np.sin(t)

t0 = 0
N = 20
h = 2 / N
y_start = np.array(
    [[y(t0), dy(t0)], [y(t0 + h), dy(t0 + h)], [y(t0 + 2 * h),
    dy(t0 + 2 * h)], [y(t0 + 3 * h), dy(t0 + 3 * h)]])
```

Подпрограмма, реализующая метод Адамса-Моултона

```
def am(f, y_start, t0, h, n):
    y = np.ndarray((n + 1, 2))
    t = np.ndarray(n + 1)
    t[0:4] = [t0, t0 + h, t0 + 2 * h, t0 + 3 * h]
    y[0:4] = y_start
    for i in range(3, n):
        t[i + 1] = t[i] + h
        y[i + 1] = y[i] + h / 24 * (
```

```
                55 * f(t[i], y[i]) - 59 * f(t[i - 1], y[i - 1]) +
37 * f(t[i - 2], y[i - 2]) - 9 * f(t[i - 3], y[i - 3]))
        y[i + 1] = y[i] + h / 24 * (9 * f(t[i + 1], y[i + 1]) + 19
* f(t[i], y[i]) - 5 * f(t[i - 1], y[i - 1]) + f(t[i - 2],y[i - 2]))
    return t, y

res = am(f, y_start, t0, h, N)
```

График решения, полученного по явному методу, имеет «пилообразную» форму, что свидетельствует о неустойчивости метода (рис 9.5)

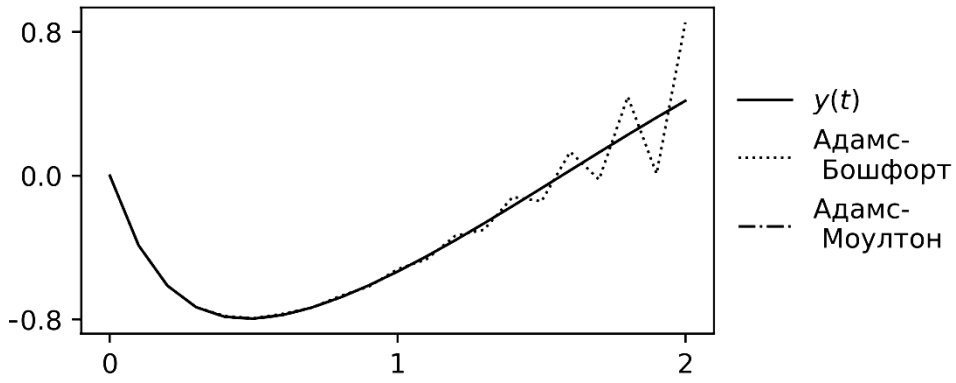

Рис. 9.5

Максимальная величина погрешности на отрезке для явного метода 0.4363, для неявного метода 0.0013

При уменьшении шага в два раза ситуация существенно улучшится (рис. 9.6)

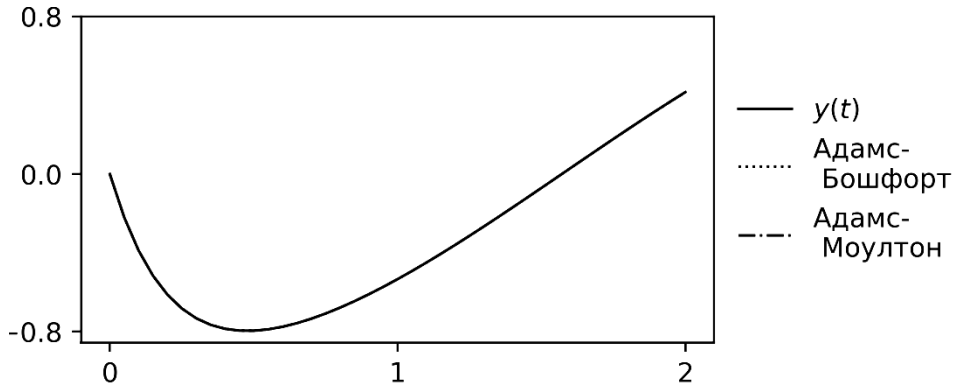

Рис. 9.6

Максимальная величина погрешности на отрезке для явного метода составляет 0.001, для неявного метода 0.00006. Заметим, что стартовые точки также следует пересчитать с учетом нового шага.

На практике довольно часто встречаются системы дифференциальных уравнений, которые принято называть жесткими.

Для их решения имеет смысл применять А-устойчивые методы, простейшим из которых является неявный метод Эйлера.

Пример 9.6. Продемонстрируем работу явного и неявного методов Эйлера на решении автономной системы размерности 2x2 с коэффициентом жесткости 39.

$$\begin{cases} y_1' = -20y_1 - 19y_2 \\ y_2' = -19y_1 - 20y_2 \\ y_1(0) = 2 \;\; y_2(0) = 0 \end{cases}$$

Точное решение имеет вид:

$$\begin{cases} y_1 = e^{-39t} + e^{-t} \\ y_2 = e^{-39t} - e^{-t} \end{cases}$$

Данную систему можно записать в матричной форме $y' = Ay$, где A – матрица коэффициентов системы.

Расчетная формула неявного метода имеет вид: $y_{i+1} = y_i + hAy_{i+1}$. Используя свойство линейности системы, неявный метод можно реализовать в виде явной формулы $y_{i+1} = \left(E - hA\right)^{-1} y_i$. Тогда можно использовать программу из примера 9.1, заменив расчетную формулу.

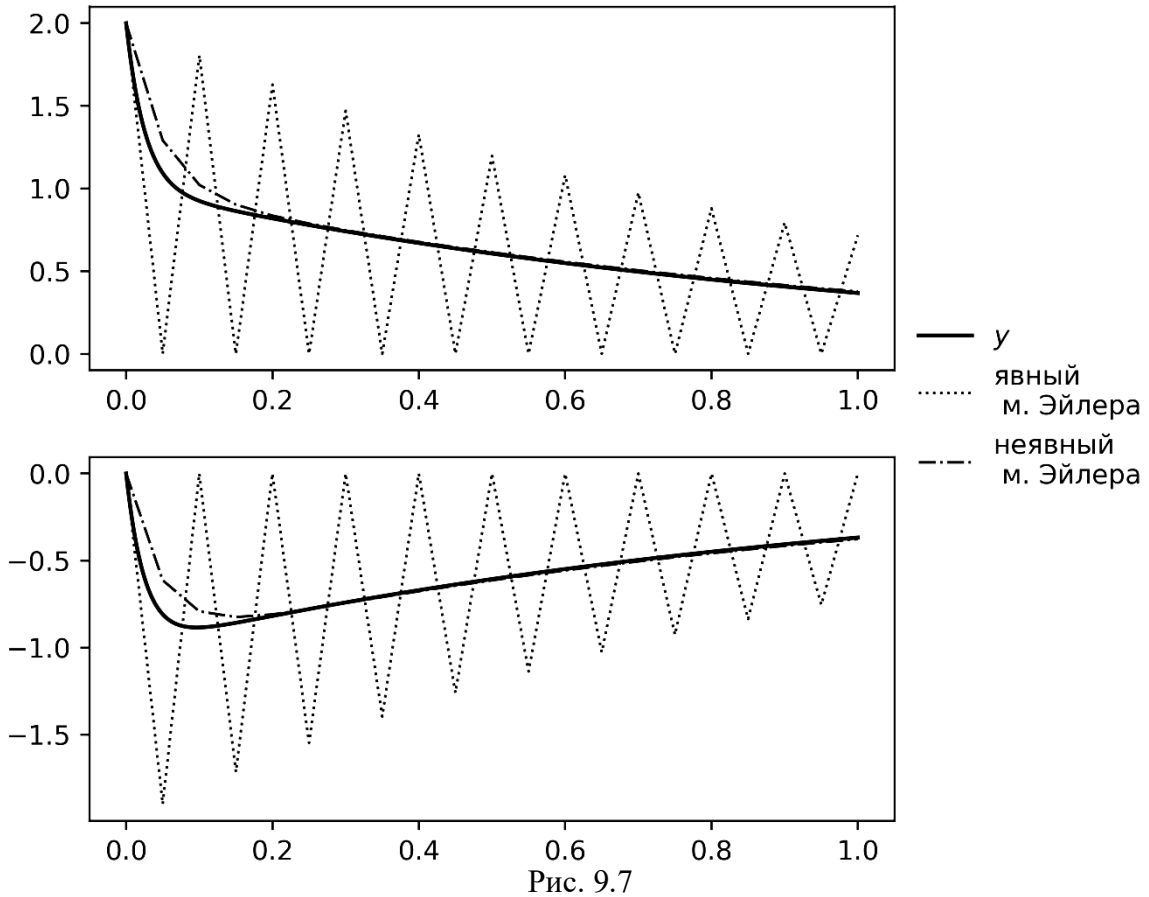

Рис. 9.7

Число жесткости системы определяется как отношение максимального и минимального собственных чисел матрицы системы (при дополнительных условиях на знак). Найдем их.

```
A = np.array([[-20, -19], [-19, -20]])
print(np.linalg.eigvals(A))
[ -1. -39.]
```

Число жесткости $S = 39$.

При выборе шага из условия устойчивости $h < 2/39$, например, $h = 0.02$ решение становится устойчивым, однако, точность вычислений невелика.

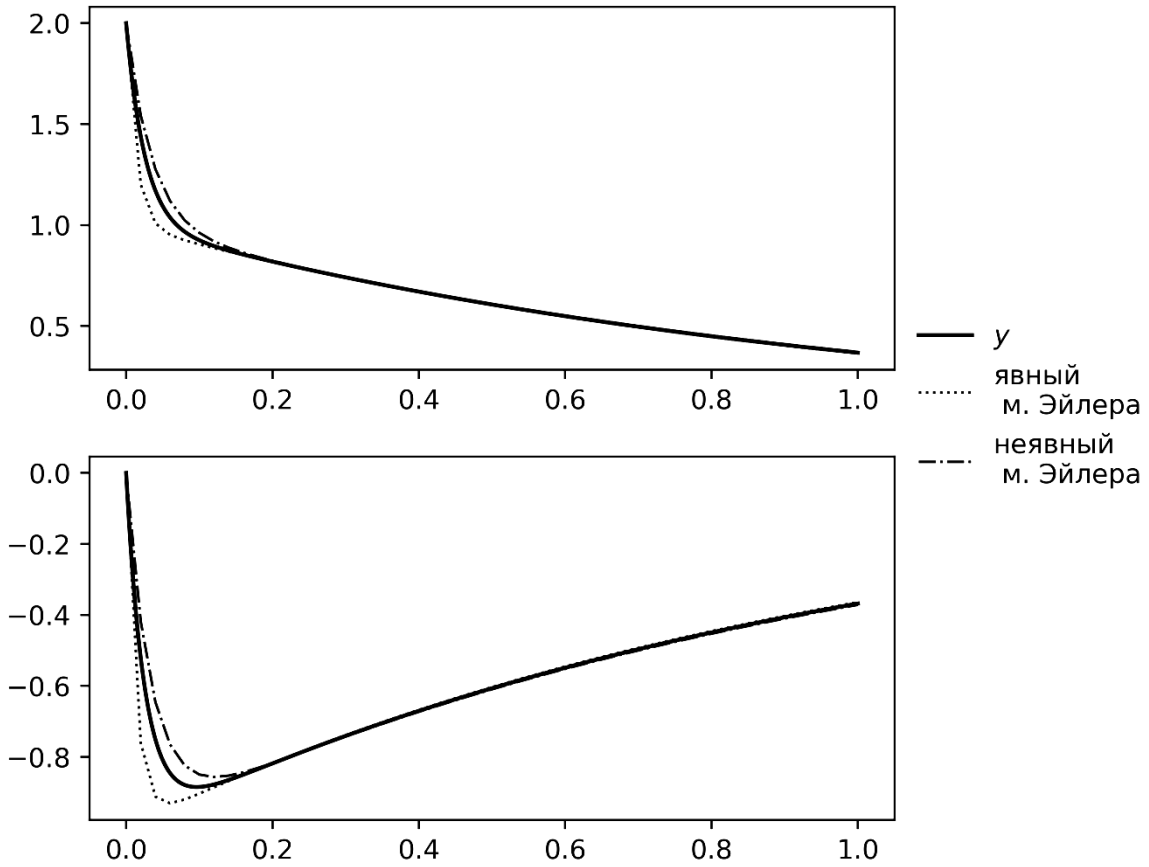

Рис. 9.8

Неявный метод обеспечивает погрешность примерно в 2 раза меньшую, чем явный.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое ‘сборка мусора’?
2. Какие существуют операторы цикла?
3. Какие виды параметров могут быть у функции?
4. Что делает оператор ***continue***?
5. Зачем нужен оператор ***break*?**
6. Как осуществляется запись в файл?
7. Как осуществляется чтение из файла?
8. Как осуществляется сериализация?
9. Как осуществляется десериализация?
10. Как создаются массивы?
11. Как выполняются операции над массивами?
12. Как задаётся цвет линии при рисовании графиков?
13. Как задаётся стиль линии при рисовании графиков?
14. Для чего производится предварительное исследование уравнения?
15. Что понимают под сходимостью итерационного метода?
16. Какие методы решения нелинейных уравнений обладают сверхлинейной скоростью сходимости?
17. Каков критерий окончания итераций в методе простой итерации?
18. Чем отличаются прямые методы решения задачи от итерационных?
19. В чем состоит алгоритм метода Гаусса решения СЛАУ ?
20. Для решения каких систем линейных уравнений эффективно использовать метод прогонки?
21. Какая библиотечная функция Python реализует метод прогонки?
22. Сформулируйте задачу глобальной интерполяции функции.
23. Какая библиотечная функция реализует интерполяцию с помощью многочлена Лагранжа?
24. Что такое интерполяционный сплайн?
25. В чем различие между интерполяцией и приближением функций в смысле наименьших квадратов?
26. Какова идея построения квадратурных формул интерполяционного типа?
27. Как практически оценивается погрешность найденного приближенного значения интеграла?
28. Можно ли вычислить тройной интеграл с помощью библиотечных функций Python?
29. Что такое одношаговый метод решения задачи Коши? Приведите пример.
30. Назовите основные различия одношаговых и многошаговых методов решения задачи Коши.

СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1. Амосов, А.А. Вычислительные методы / А.А. Амосов, Ю.А. Дубинский, Н.В. Копченова. – М.: Издательский дом МЭИ, 2008. – 672 с.
2. Вабищевич, П.Н. Численные методы: вычислительный практикум. Практическое применение численных методов при использовании алгоритмического языка Python / П.Н. Вабищевич. – М.: ЛЕНАНД, 2018. – 320 с.
3. Князев, А.В. Основы программирования на языке Python / А.В. Князев – М.: Издательство МЭИ, 2021. – 76 с.
4. Мартелли, А. Python. Справочник. Полное описание языка: пер. с англ. / А. Мартелли, А. Рейвенскрофт, С. Холден. – СПб.: ООО “Диалектика”, 2019. – 896 с.
5. Функции пакета scipy. Документация. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root_scalar.html#scipy.optimize.root_scalar

Учебное издание

Амосова Ольга Алексеевна
Вестфальский Алексей Евгеньевич
Князев Анатолий Васильевич
Крымов Никита Евгеньевич

ЧИСЛЕННЫЕ МЕТОДЫ НА ЯЗЫКЕ PYTHON

Учебное пособие

Редактор издательства
Компьютерная вёрстка

Подписано в печать		Печать офсетная	Формат
Физ. печ.л.	Тираж	Изд.	Заказ

Оригинал-макет подготовлен в РИО НИУ «МЭИ»
111250, Москва, ул. Красноказарменная, д.14
Отпечатано в типографии НИ У «МЭИ»
111250, Москва, ул. Красноказарменная, д.13