Министерство науки и высшего образования Российской Федерации
Казанский национальный исследовательский
технологический университет

А. Н. Титов, Р. Ф. Тазиева

РЕШЕНИЕ ЗАДАЧ ЛИНЕЙНОЙ АЛГЕБРЫ И ПРИКЛАДНОЙ МАТЕМАТИКИ В PYTHON

РАБОТА С БИБЛИОТЕКОЙ SCIPY

Учебно-методическое пособие

Казань
Издательство КНИТУ
2023

УДК 512.64:004(075)
ББК 22.143:32.97я7
Т45

Печатается по решению редакционно-издательского совета Казанского национального исследовательского технологического университета

Рецензенты:
д-р пед. наук, доц. Ю. В. Торкунова
канд. экон. наук, доц. О. С. Семичева

Титов А. Н.

Т45 Решение задач линейной алгебры и прикладной математики в Python. Работа с библиотекой SciPy : учебно-методическое пособие / А. Н. Титов, Р. Ф. Тазиева; Минобрнауки России, Казан. нац. исслед. технол. ун-т. – Казань : Изд-во КНИТУ, 2023. – 124 с.

ISBN 978-5-7882-3319-2

Рассмотрены задачи по линейной алгебре, вычислительной и прикладной математике, информационным технологиям и их решения с использованием языка программирования Python. Описана технология работы с библиотекой SciPy, приведены необходимые теоретические сведения и формулы для решения рассмотренных задач. Для оценки уровня усвоения студентами пройденного материала предложены варианты заданий для самостоятельной работы.

Предназначено для бакалавров направлений подготовки 28.03.02 «Наноинженерия», 09.03.02 «Информационные системы и технологии», изучающих дисциплины «Обработка экспериментальных данных», «Прикладная математика», «Вычислительная математика», «Информационные технологии».

Подготовлено на кафедре информатики и прикладной математики.

УДК 512.64:004(075)
ББК 22.143:32.97я7

ISBN 978-5-7882-3319-2

С О Д Е Р Ж А Н И Е

ВВЕДЕНИЕ

Python является одним из наиболее востребованных языков программирования для обработки данных. Во многом это обусловлено тем, что, несмотря на свою относительную простоту, он является в то же время и очень мощным средством, которое позволяет решать широкий круг возникающих на практике задач. Кроме того, для анализа и обработки данных в нем имеется достаточно много открытых библиотек. Одной из таких библиотек является библиотека SciPy. SciPy – это библиотека Python с открытым исходным кодом, предназначенная для решения научных, инженерных и математических задач. Она построена на базе библиотеки NumPy и позволяет управлять данными, а также визуализировать их. SciPy содержит набор модулей для обработки результатов научных исследований. Среди этих модулей – модуль cluster, содержащий реализацию алгоритмов кластерного анализа, модуль constans, содержащий физические и математические константы, fftpack для быстрого преобразования Фурье, integrate для решения задач численного интегрирования и решения дифференциальных уравнений, interpolate для решения задач интерполяции, linalg для решения задач линейной алгебры, optimize для оптимизации и численного решения уравнений и систем уравнений, в том числе нелинейных, signal для обработки сигналов, stats для решения задач теории вероятностей и математической статистики и многие другие. Подробное описание можно найти в официальной документации.

В рамках данного учебного пособия авторы поставили перед собой цель: собрать имеющуюся информацию по работе в SciPy, переработать ее с учетом тех типов задач, которые решаются в преподаваемых авторами дисциплинах. Авторы надеются облегчить восприятие изложенного материала, снабдив пособие большим количеством примеров и пояснений к ним.

В данном пособии рассмотрены вопросы работы с некоторыми функциями модулей linalg, optimize, integrate и interpolate.

Пособие включает в себя 6 глав. В первой главе показано, как с помощью функций библиотеки SciPy можно решать системы линейных уравнений, вычислять определители и находить обратные матрицы, вычислять собственные значения и нормированные собственные вектора матрицы, находить произведение матриц. Во второй главе приведены примеры решения трансцендентных уравнений двумя

численными методами и показано, как решаются алгебраические, трансцендентные уравнения и системы линейных и нелинейных уравнений с помощью функций root_scalar(), root() и fsolve(). Рассмотрены вопросы выбора начального приближения для решения систем двух нелинейных уравнений с использованием функции plot_implicit. В третьей главе рассмотрены способы решения задач одномерной аппроксимации полиномами и другими функциями с использованием функций lstsq(), curve_fit() и leastsq(). В четвертой главе приведено большое количество примеров решения задач интерполяции с использованием интерполяционных полиномов, кубических сплайнов и *B*-сплайнов (функции curve_fit(), leastsq(), interp1d(), splrep(), splrep()). В пятой главе рассмотрены задачи численного интегрирования функций, заданных таблично и аналитически (функции trapz(), cumtrapz(), quad(), fixed_quad(), romberg()). В шестой главе приведены примеры решения ОДУ первого порядка и систем таких уравнений как численно (функции odeodeint() и solve_ivp()), так и аналитически (функция dsolve() библиотеки SymPy).

Материал, изложенный в пособии, может быть использован при проведении лабораторных занятий по обработке экспериментальных данных, линейной алгебре, вычислительной и прикладной математике, информационным технологиям.

1. РЕШЕНИЕ ЗАДАЧ ЛИНЕЙНОЙ АЛГЕБРЫ В БИБЛИОТЕКЕ SCIPY

Для решения задач линейной алгебры в SciPy предназначен модуль scipy.linalg. С его помощью можно решать системы линейных уравнений, а также производить различные действия над матрицами: вычислять определитель, строить обратную матрицу, вычислять собственные значения и нормированные собственные вектора матрицы, вычислять произведения матриц и многое другое. Пакет scipy.linalg может стать полезным инструментом для решения транспортной задачи, балансировки химических уравнений и электрических нагрузок, полиномиальной интерполяции и т. д. В официальной документации сказано, что scipy.linalg содержит все функции numpy.linalg библиотеки NumPy, а также дополнительные функции, не входящие в numpy.linalg.

1.1. Установка библиотеки

Для работы с библиотекой ее нужно установить на компьютер. Такие среды разработки Python, как Jupyter Notebook, Google Colab, уже включают в себя различные библиотеки, в том числе и SciPy. Их не надо устанавливать, а нужно только импортировать. Если предполагается работа в среде, в которой нет библиотеки SciPy, ее необходимо установить. Для установки библиотеки (модуля) понадобится пакетный менеджер Python pip. Pip (Package installer for Python) – это система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Прежде всего нужно убедиться, что сам pip установлен. Начиная с версии Python 3.4 pip поставляется вместе с интерпретатором Python. После установки pip сначала проверяем, не установлена ли уже данная библиотека. В командной строке необходимо набрать *pip freeze*. Получится список, аналогичный приведенному на рис. 1.1.

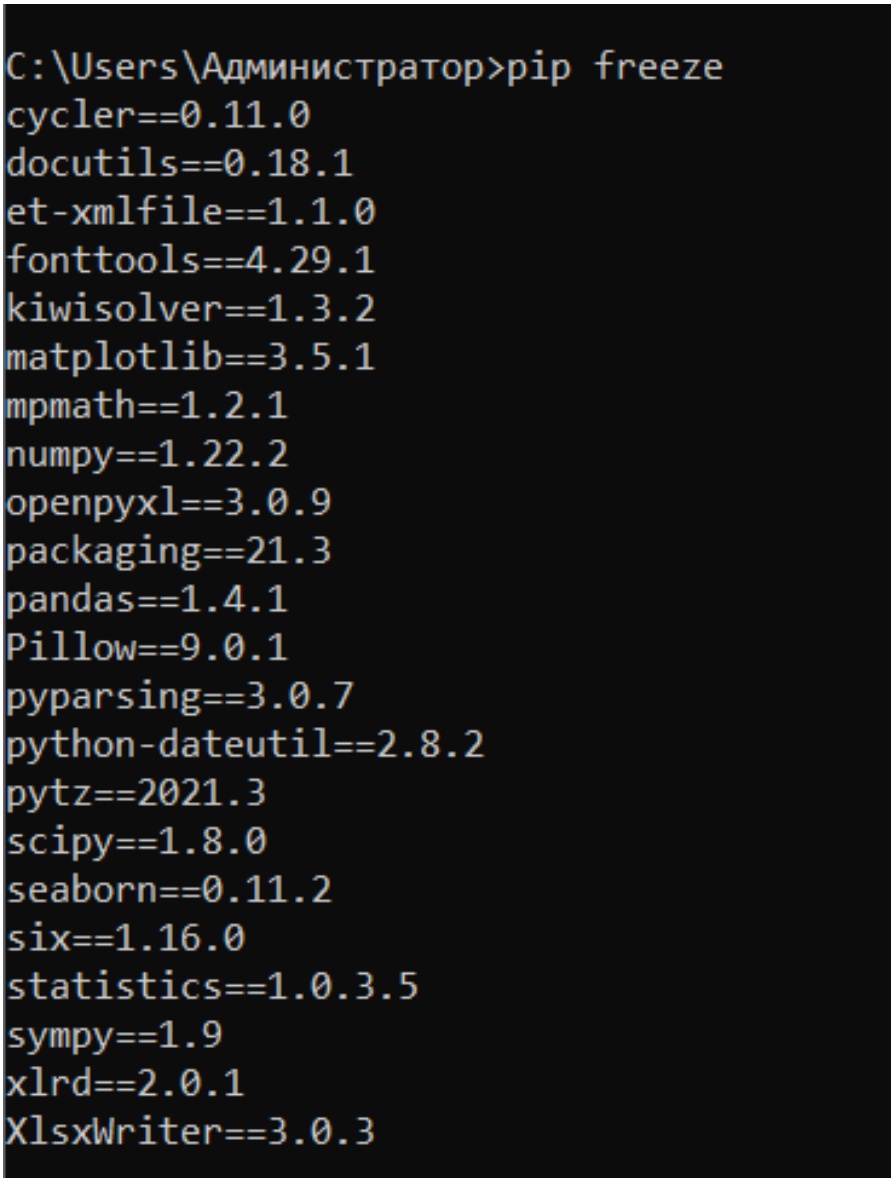

Рис. 1.1. Список установленных библиотек Python

Из приведенного списка видно, что библиотека уже установлена и ее версия – 1.8.0. Если бы в списке отсутствовала библиотека SciPy, загрузить ее можно было командой

pip install scipy

По умолчанию установится последняя версия библиотеки. Узнать версию, если вы работаете в среде Google Colab или в другой среде разработки, можно, выполнив команды

import scipy; scipy.__version__

Версия авторов пособия не является последней. Ее номер '1.4.1'.

1.2. Работа с модулем scipy.linalg

Для решения задач линейной алгебры в SciPy предназначен модуль scipy.linalg. Продемонстрируем некоторые возможности модуля на примерах.

Пример 1.1. Решить систему линейных уравнений

$$\begin{cases} 3x_1 - 2x_2 + 4x_3 = 8 \\ -x_1 + 12x_3 = 18 \\ 4x_1 + 3x_2 - x_3 = 7 \end{cases}.$$

Решение. Для большей наглядности перепишем систему уравнений в матричном виде $Ax=b$:

$$\underbrace{\begin{pmatrix} 3 & -2 & 4 \\ -1 & 0 & 12 \\ 4 & 3 & -1 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 8 \\ 18 \\ 7 \end{pmatrix}}_{b}.$$

Функция, с помощью которой можно решить систему линейных уравнений в SciPy, – linalg.solve(), входными данными для которой является матрица A и вектор b. Их нужно задать в виде двух массивов: A – массив 3 на 3 и b – массив 3 на 1. Вводим их как массивы np.array. Код программы следующий:

```
import numpy as np
from scipy.linalg import *
A = np.array([
    [3,-2,4],
    [-1,0,12],
    [4,3,-1]
])
b = np.array([8,18,7]).reshape(3,1)
x=solve(A,b)
for i in range(len(b)):
  print("x(%i)=%.7f" % (i+1,x[i]))
```

Результат работы программы:

```
x(1)=1.2897196
x(2)=1.1495327
x(3)=1.6074766
```

Комментарий к коду. Импортируем библиотеку NumPy. Она нужна для создания массивов с данными. Из библиотеки SciPy импортируем все функции – знак «*», хотя в данном случае нам нужна лишь одна функция – solve(). Это можно было сделать следующим образом: from scipy.linalg import solve. Вводим элементы матрицы *A*. Матрицу *A* можно было задать списком. Создаем вектор-строку *b* как массив с именем *b*. Чтобы сделать его вектор-столбцом, вызываем метод reshape(). Решаем систему уравнений с помощью функции solve(), передав в нее в качестве аргументов матрицу *A* и вектор-столбец *b*. После этого вместо вывода результатов в цикле достаточно было просто записать: *x*. В этом случае был бы получен ответ в виде

```
array([[1.28971963],
       [1.14953271],
       [1.60747664]])
```

В коде приведен другой вариант вывода. В цикле по переменной *i*, которая принимает значения 0, 1, 2 (len(b)=3, так как вектор содержит 3 элемента, а нумерация ведется с 0), выводятся значения x_i. Индексы у переменной *x* увеличены на 1, чтобы они совпадали с индексами в условии примера.

Попытка решить систему, имеющую бесчисленное множество решений, например, такую: $\begin{cases} 2x_1 + 3x_2 = 5 \\ 4x_1 + 6x_2 = 10 \end{cases}$, приводит к ошибке и выдаче сообщения: LinAlgError: Matrix is singular (т. е. определитель матрицы равен нулю), хотя некоторые из программных продуктов выдают в этом случае одно из возможных решений (например, Scilab). То же сообщение выдается, если система уравнений несовместна.

Далее рассмотрим, как можно найти обратную матрицу.

Пример 1.2. Вычислить обратную матрицу для матрицы $A = \begin{pmatrix} 3 & 6 & 7 \\ 2 & 3 & 1 \\ 6 & 5 & 4 \end{pmatrix}$.

Решение. Матрица, обратная для матрицы A, – это такая матрица B, что $A \cdot B = E$, где E – это единичная матрица, состоящая из единиц по диагонали. Остальные элементы матрицы E равны 0. Код программы:

```
from scipy.linalg import *
A = [
   [3,6,7],
   [2,3,1],
   [6,5,4]
]
B = inv(A)
B
```

В результате выполнения программы получаем

```
array([[-0.14893617, -0.23404255,  0.31914894],
       [ 0.04255319,  0.63829787, -0.23404255],
       [ 0.17021277, -0.44680851,  0.06382979]])
```

Проверить полученный результат можно методом .dot, но он не поддерживается при работе со списками. Проверим правильность полученного результата с помощью кода

```
import numpy as np
from scipy.linalg import *
A = np.array([
   [3,6,7],
   [2,3,1],
   [6,5,4]
])
B = inv(A)
A.dot(B) # произведение матриц A и B
```

Результат перемножения матриц A и B – единичная матрица:

```
array([[ 1.00000000e+00,  1.11022302e-16, -1.38777878e-17],
       [-5.55111512e-17,  1.00000000e+00, -4.16333634e-17],
       [-1.11022302e-16,  4.44089210e-16,  1.00000000e+00]])
```

Вычислим определитель матрицы.

Пример 1.3. Найти определитель матрицы $A = \begin{pmatrix} 3 & 6 & 7 \\ 2 & 3 & 1 \\ 6 & 5 & 4 \end{pmatrix}$.

Решение. Для вычисления определителя в библиотеке используют функцию linalg.det(). Код программы вычисления определителя следующий:

```
import numpy as np
from scipy import linalg
A = np.array([
   [3,6,7],
   [2,3,1],
   [6,5,4]])
B = linalg.det(A)
print(B)
```

Результат: −47.0.

В модуле предусмотрены функции для поиска собственных векторов и собственных значений матрицы.

Собственные значения матрицы находят, решив матричное уравнение det(A-λE)=0. Собственный вектор матрицы – это вектор, умножение матрицы на который дает тот же вектор, умноженный на некоторое число: $AN=\lambda N$, где A – матрица; N – собственный вектор; λ – некоторое число. Для вычисления собственных значений матрицы и нормированных собственных векторов в модуле linalg применяют функцию eig().

Пример 1.4. Найти собственные значения и нормированные собственные вектора матрицы $\begin{pmatrix} -1 & -6 \\ 2 & 6 \end{pmatrix}$.

Решение:

```
import numpy as np; from scipy import linalg
A = np.array([[-1, -6], [2, 6]]); linalg.eig(A)
```

Результат работы программы:

(array([2.+0.j, 3.+0.j]), array([[-0.89442719, 0.83205029], [0.4472136 , -0.5547002]])).

Пояснение. Собственные значения матрицы A содержатся в первом массиве: $\lambda_1 = 2$, $\lambda_2 = 3$. Проверим это:

$$\begin{vmatrix} -1-2 & -6 \\ 2 & 6-2 \end{vmatrix} = \begin{vmatrix} -3 & -6 \\ 2 & 4 \end{vmatrix} = -12+12 = 0,$$

$$\begin{vmatrix} -1-3 & -6 \\ 2 & 6-3 \end{vmatrix} = \begin{vmatrix} -4 & -6 \\ 2 & 3 \end{vmatrix} = -12+12 = 0.$$

Собственными векторами матрицы являются вектора $\begin{pmatrix} -2 \\ 1 \end{pmatrix}$ и вектор $\begin{pmatrix} \frac{3}{2} \\ -1 \end{pmatrix}$, так как $\begin{pmatrix} -1 & -6 \\ 2 & 6 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 \\ 2 \end{pmatrix} = 2\begin{pmatrix} -2 \\ 1 \end{pmatrix}$ и $\begin{pmatrix} -1 & -6 \\ 2 & 6 \end{pmatrix} \cdot \begin{pmatrix} \frac{3}{2} \\ -1 \end{pmatrix} = \begin{pmatrix} \frac{9}{2} \\ -3 \end{pmatrix} = 3\begin{pmatrix} \frac{3}{2} \\ -1 \end{pmatrix}$. Пронормируем собственные вектора: $\begin{pmatrix} \frac{-2}{\sqrt{1^2 + (-2)^2}} \\ \frac{1}{\sqrt{1^2 + (-2)^2}} \end{pmatrix} = \begin{pmatrix} \frac{-2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{pmatrix} = \begin{pmatrix} -0.8944271909999159 \\ 0.4472135954999579 \end{pmatrix}$.

$\begin{pmatrix} \frac{\frac{3}{2}}{\sqrt{\left(\frac{3}{2}\right)^2 + (-1)^2}} \\ \frac{-1}{\sqrt{\left(\frac{3}{2}\right)^2 + (-1)^2}} \end{pmatrix} = \begin{pmatrix} \frac{\frac{3}{2}}{\sqrt{3.25}} \\ \frac{-1}{\sqrt{3.25}} \end{pmatrix} = \begin{pmatrix} 0.8320502943378437 \\ -0.5547001962252291 \end{pmatrix}$. Это соответственно первый и второй столбцы второго массива, полученного в результате работы программы. Если в коде решения примера 1.4 поменять строку, в которой вызывалась функция eig(), на строку la: v = linalg.eig(A), то *la* – это массив с собственными значениями матрицы *A*, а *v* – матрица, содержащая собственные вектора.

SciPy предоставляет несколько функций для создания специальных матриц, которые часто используются в научных исследованиях. В этом же модуле можно с помощью функции linalg.lstsq() решить задачу аппроксимации методом наименьших квадратов.

Задания для самостоятельной работы

В приведенных заданиях α – это номер института, β – последняя цифра года поступления, γ и θ – две последние цифры номера группы,

μ – первая цифра номера студента по списку, ν – последняя цифра номера по списку. Так, у пятого студента группы 723112 $\alpha = 7$, $\beta = 3$, $\gamma = 1$, $\theta = 2$, $\mu = 0$, $\nu = 5$.

Задание 1. Перемножить матрицы *A*, *B* и *C*:

$$A = \begin{pmatrix} 2 & 3+\alpha & 4-\mu \\ \beta & \gamma & \nu \\ 5 & 10\mu & -2 \end{pmatrix}, B = \begin{pmatrix} \beta & -3 & 4+\mu \\ \alpha & 4 & \nu \\ 5\mu & 10 & -2 \end{pmatrix}, C = \begin{pmatrix} -1 & 3+\mu & 4-\nu \\ 2 & 1 & \nu \\ 5 & 5-\mu & -2 \end{pmatrix}.$$

Задание 2. Вычислить определитель, вектор собственных значений матрицы *G* и матрицу, обратную ей: $G = \begin{pmatrix} 2 & 3+\alpha & 4+\mu & 6 \\ \beta+1 & \gamma & \nu & 12 \\ 5 & -2 & 16 & \mu+1 \\ 2 & 7 & 11 & -6 \end{pmatrix}$.

Задание 3. Известно, что в результате умножения матрицы *A* на матрицу *B* получена матрица *C*. Найти матрицу *B*, если

$$A = \begin{pmatrix} 2 & -5 & 7 \\ 8 & 1 & 9 \\ 3. & -4 & 12 \end{pmatrix}, C = \begin{pmatrix} -25 & -17 & 78 \\ 11 & 89 & 76 \\ -5 & 24 & 115 \end{pmatrix}.$$

Задание 4. Решить систему линейных уравнений

$$\begin{cases} 3x_1 + x_2 + x_3 + \gamma x_4 = \alpha \\ x_1 - \mu x_2 + \nu x_3 + 4x_4 = \beta \\ -5x_1 - x_3 - 7x_4 = -5 \\ x_1 - 6x_2 + \alpha x_3 + 6x_4 = 3 \end{cases}.$$

2. ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ УРАВНЕНИЙ И ИХ РЕАЛИЗАЦИЯ В БИБЛИОТЕКЕ SCIPY

Уравнения можно поделить на два типа: алгебраические и трансцендентные. Уравнение называется алгебраическим, если его можно представить в виде $\sum_{i=0}^{n} a_i x^i = 0$. Это каноническая форма записи алгебраического уравнения. Пример алгебраического уравнения:

$$2.5x^5 - 4x^4 - 3x^2 - 8x + 12 = 0.$$

Если левая часть уравнения $f(x) = 0$ не является алгебраической функцией, то уравнение называется трансцендентным. Пример трансцендентного уравнения:

$$x \ln x + x^2 \cos x - \frac{x}{\lg x + 3} - 8 = 0.$$

Решить уравнение означает найти такие значения x, при которых уравнение $f(x) = 0$ превращается в тождество.

Известно, что алгебраическое уравнение имеет ровно n корней – вещественных или комплексных. Если $n = 1, 2, 3$, а иногда и 4 (биквадратное уравнение), то существуют точные методы решения алгебраических уравнений. Если же $n > 4$ или уравнение трансцендентное, то таких методов, как правило, не существует, и решение уравнения находят приближенными методами. Всюду при дальнейшем изложении будем предполагать, что $f(x)$ – непрерывная на интервале поиска корня функция. Методы, которые предлагает SciPy, пригодны для поиска некратных (т. е. изолированных) корней.

Существует множество методов численного решения уравнений. Это обусловлено отсутствием универсального метода, пригодного для решения любых уравнений с устраивающей всех скоростью сходимости к корню. У каждого метода есть свои достоинства и недостатки. У многих методов необходимо проверять условия сходимости, т. е. условия, гарантирующие, что за конечное число шагов применения данного метода мы получим корень с любой наперед заданной погрешностью ε.

Решение уравнения численными методами состоит из двух этапов: первый этап – отделение корня, второй – его уточнение.

Отделить корень – значит указать такой отрезок $[a, b]$, на котором содержится *ровно один* корень уравнения $f(x) = 0$.

На рис. 2.1 представлен график функции, имеющей несколько корней. В качестве отрезков, на которых корень отделен, можно выбрать, например, такие: $[-3, -2]$, $[-1, 0]$, $[1, 2]$, $[3, 4]$.

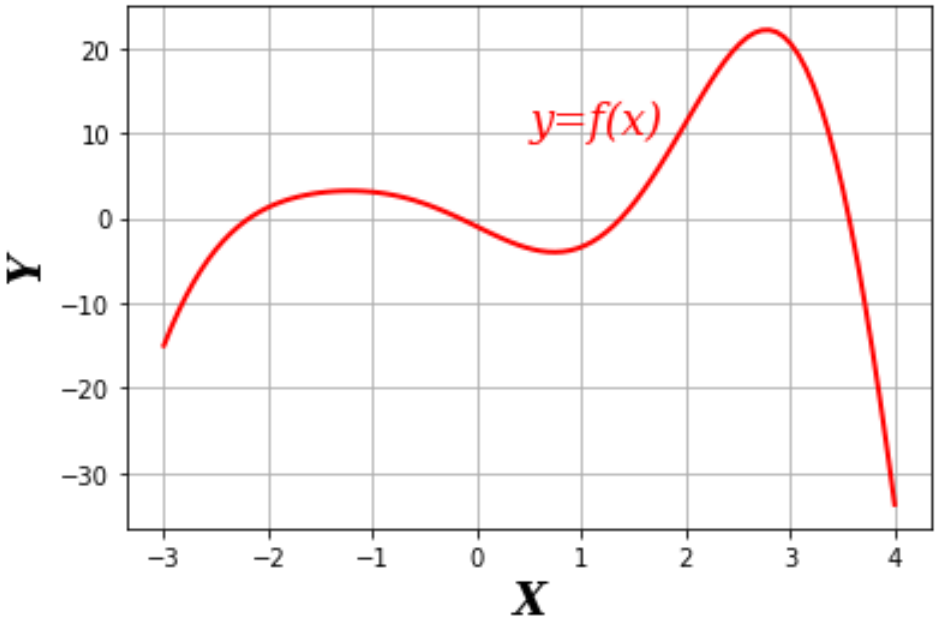

Рис. 2.1. График функции, имеющей несколько корней на отрезке [−2, 4]

Корень можно отделить аналитически и графически.

Не существует алгоритмов отделения корня, пригодных для любых функций $f(x)$. Если удастся подобрать такие a и b, что

1) $f(a)\,f(b) < 0$,
2) $f(x)$ – непрерывная на $[a, b]$ функция,
3) $f(x)$ – монотонная на $[a, b]$ функция,

то можно утверждать, что на отрезке $[a, b]$ корень отделен.

Условия 1–3 – достаточные условия отделения корня, т. е. если эти условия выполняются, то корень отделен, но невыполнение, например, условий 2 или 3 не всегда означает, что корень не отделен. Предложенный метод поиска отрезка $[a, b]$ – это аналитический способ отделения корня. Корень можно отделить и графически. Графические

возможности библиотек Python позволяют достаточно просто решить эту задачу.

Второй этап решения уравнения – уточнение корня. Уточнить корень – значит найти его приближенное значение с заданной погрешностью ε. Этот этап реализован в нескольких функциях библиотеки.

2.1. Уточнение корня уравнения. Функция root_scalar()

Функция root_scalar() позволяет решить уравнение одним из 8 предусмотренных в ней методов. Каждый метод имеет разный набор обязательных параметров. Синтаксис функции:

root_scalar(f, args=(), method=None, bracket=None, fprime=None, fprime2=None, x0=None, x1=None, xtol=None, rtol=None, maxiter=None, options=None).

Аргументы функции root_scalar():

f – вызываемая функция для поиска корня. Можно определить ее так, чтобы при необходимости возвращались и функция *f*, и ее производные;

args – кортеж, задающий дополнительные аргументы, передаваемые целевой функции и ее производным, если такие имеются;

method – метод, символьная строка, задающая способ решения уравнений. Тип решателя. Принимает одно из значений: {'bisect', 'brentq', 'brenth', 'ridder', 'toms748', 'newton', 'secant', 'halley'}. Необязательный параметр;

bracket – последовательность из двух чисел типа float, необязательный параметр. Интервал поиска корня. Функция *f* должна иметь разные знаки на концах заданного интервала;

x0 – переменная типа float, начальное приближение;

x1 – переменная типа float, второе начальное приближение;

fprime – переменная типа bool или callable, необязательный параметр. Если *fprime* является булевской переменной со значением True, предполагается, что функция *f* определена так, что возвращает и значение *f*, и значение ее производной. *fprime* также может быть именем

функции, возвращающей производную от *f*. В этом случае функция должна содержать те же аргументы, что и функция *f*;

fprime2 – переменная типа bool или callable, необязательный параметр. Если *fprime2* является булевской переменной со значением True, предполагается, что функция *f* определена так, что возвращает и значение *f*, и значение ее первой и второй производных. *fprime2* также может быть именем функции, возвращающей вторую производную от *f*. В этом случае функция должна содержать те же аргументы, что и функция *f*;

xtol и *rtol* – аргументы, имеющие тип float и задающие желаемые абсолютную и относительную погрешность результата. Вычисления прекращаются при достижении этой погрешности;

maxiter – переменная, имеющая тип int и задающая максимальное количество итераций.

Атрибутами решения являются: *converged* – логический флаг, указывающий, успешно ли завершен алгоритм; *flag* – переменная, описывающая причину завершения; *function_calls* – переменная, показывающая количество вызовов функции *f*; *iterations* – переменная, показывающая, сколько итераций сделано для достижения заданной погрешности; *root* – корень уравнения.

По умолчанию используется лучший метод, доступный при решении конкретной задачи. Если задан интервал поиска, то может быть использован один из методов, для которых требуется отрезок, на котором содержится корень (например, метод деления отрезка пополам). Если указаны производная и начальное значение, можно выбрать один из методов, основанных на использовании производной (например, метод касательных). Если ни один метод не будет сочтен применимым, он сгенерирует исключение.

Рассмотрим работу нескольких из предусмотренных в функции методов поиска корня уравнения.

Метод деления отрезка пополам. Самый простой метод, пригодный для поиска корней любых непрерывных функций, – *метод деления отрезка пополам*.

Предположим, что отрезок $[a, b]$, на котором отделен корень уравнения, уже найден. Пусть, например, $f(a) < 0$, $f(b) > 0$ (рис. 2.2). Буквой ξ обозначен корень уравнения.

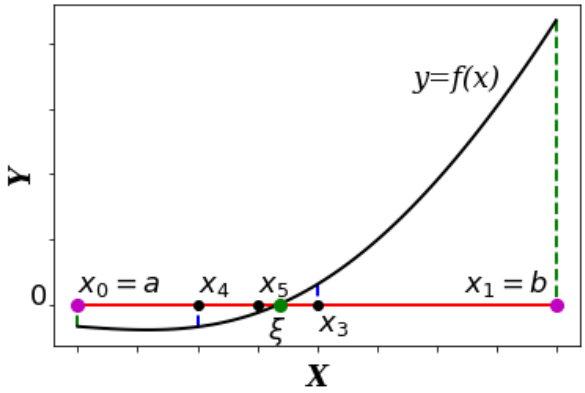

Рис. 2.2. Геометрическая интерпретация метода деления отрезка пополам

Обозначим $x_0 = a$, $x_1 = b$ и вычислим точку $x_3 = \frac{x_0 + x_1}{2}$. Если вместо корня взять точку x_3, то погрешность, которую мы при этом допустим, не превысит величины $\varepsilon_1 = \frac{b-a}{2}$. Если эта погрешность не превышает заданную погрешность ε, с которой нужно уточнить корень уравнения, то вычисления прекращаем и можно считать, что корень $\xi = x_3 \pm \frac{b-a}{2}$. В противном случае определяем новый отрезок $[a, b]$, на котором отделен корень нашего уравнения. Для этого нам нужно знать знак функции в точке x_3. Из рис. 2.2 видно, что $f(x_3) > 0$. Новый отрезок – отрезок $[x_0, x_3]$, так как на концах этого отрезка функция имеет разные знаки. Повторяем процедуру для нового отрезка $[x_0, x_3]$. Вычисляем точку $x_4 = \frac{x_0 + x_3}{2}$. Если в качестве корня взять точку x_4, то можно записать приближенное значение корня: $\xi = x_4 \pm \frac{b-a}{2^2}$. Если погрешность $\varepsilon_2 = \frac{b-a}{2^2} > \varepsilon$, делим отрезок $[x_4, x_3]$ пополам, так как на концах этого отрезка функция имеет разные знаки, и вычисляем $x_5 = \frac{x_4 + x_3}{2}$. Процедура продолжается до тех пор, пока достигнутая погрешность не станет меньше заданной. После каждой итерации погрешность уменьшается в два раза.

Работу метода рассмотрим на примере.

Пример 2.1. Отделить и уточнить корень уравнения $x^3 \ln(x) - 6x\sin(x) - 12 = 0$ методом деления отрезка пополам.

Решение. Отделим корень уравнения графически. Подключаем одну из библиотек (например, Matplotlib) и строим график:

```
import numpy as np; import matplotlib.pyplot as plt
x = np.linspace(1, 3, 100);
def f(x):
   return x**3*np.log(x) - 6*x*np.sin(x)-12
plt.title(r"$График \ функции \ y \ =x^3log(x) - 6xsin(x)-12$",
        fontsize=14)
plt.xlabel("X",color='k',fontstyle="italic",
   fontsize=18,fontfamily = 'serif') # ось абсцисс
plt.ylabel("Y",color='k',fontstyle="italic",
   fontsize=18,fontfamily = 'serif') # ось ординат
plt.grid()     # включение отображение сетки
plt.plot(x, f(x),linewidth=3,c='red');  plt.show()
```

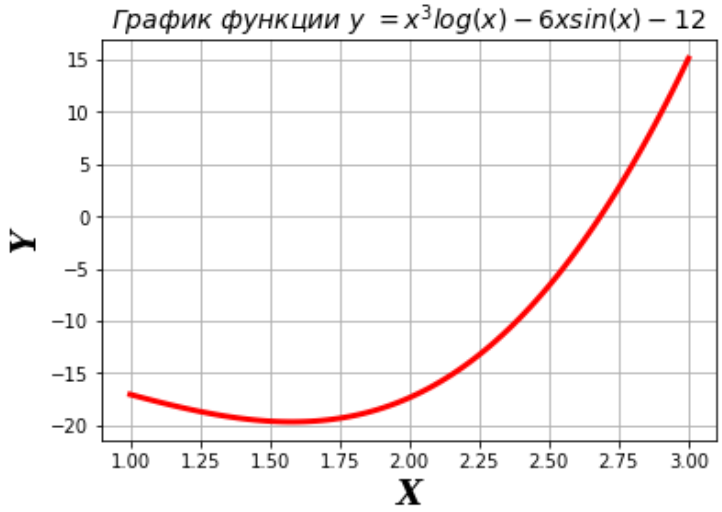

Рис. 2.3. Отделение корня уравнения $x^3ln(x) - 6xsin(x) - 12=0$

Из графика видно, что в качестве отрезка, на котором отделен корень уравнения $x^3 \ln(x) - 6x\sin(x) - 12 = 0$, можно выбрать отрезок [2.5, 2.75].

Комментарий к программе. Подключаем библиотеку NumPy для того, чтобы воспользоваться функциями этой библиотеки sin(x), log(x) и linspace(). Задаем значения *x* из интервала [1, 3]. Концы отрезка выбраны случайным образом. Если бы на выбранном отрезке не оказалось корня, эти значения можно было бы поменять. График будет строиться

по 100 точкам (последний аргумент функции linspace()). Определяем функцию $f(x)$ – левую часть нашего уравнения. Текст заголовка графика для красоты задаем стилем LaTex. Задаем стили меток по осям *X* и *Y*: текст надписи, цвет (черный), стиль и размер кегля. Наносим на график сетку. Строим график функции $f(x) = x^3 \ln(x) - 6x\sin(x) - 12$ линией красного цвета толщиной 3. Выводим график в отдельное окно.

Переходим ко второму этапу – уточнению корня. Для уточнения корня выбран отрезок [2, 3], а не [2.5, 2.75], чтобы увеличить необходимое число итераций. Результаты решения приведены в табл. 2.1.

Таблица 2.1

Решение уравнения методом деления отрезка пополам

n	a	x=(a+b)/2	b	$f(a)$	$f(x)$	$f(b)$	ε=(b-a)/2
1	2	2,5	3	-17,3664	-6,66	15,1224	0,5
2	2,5	2,75	3	-6,66004	2,74073	15,1224	0,25
3	2,5	2,625	2,75	-6,66004	-2,323	2,74073	0,125
4	2,625	2,6875	2,75	-2,32297	0,11664	2,74073	0,0625
5	2,625	2,65625	2,6875	-2,32297	-1,1261	0,11664	0,03125

Краткий комментарий. В табл. 2.1 $f(x) = x^3 \ln(x) - 6x\sin(x) - 12$. Первоначальный отрезок [2, 3] делим пополам. Получаем точку x = 2,5. Погрешность, которую мы можем гарантировать, если взять точку x в качестве корня, не превышает 0.5: ξ = 2,5 ± 0.5. На второй итерации делим пополам новый отрезок [a, b]=[2,5 3], так как на концах этого отрезка функция $f(x)$ имеет разные знаки: f(2,5) = -6,66004, f(3) = 15,1224. Делаем 5 итераций, так как после 5-й итерации погрешность не превышает величины ε = 0,03125, которая меньше заданной в условии погрешности 0,05. Вывод: корень уравнения ξ = 2,65625 ± 0,03125.

Если на отрезке [a, b] функция $f(x)$ имеет разные знаки, но корень не отделен, то, применяя процедуру метода деления отрезка пополам, найдем один из корней уравнения $f(x) = 0$. К недостаткам метода можно отнести его медленную сходимость.

Реализация этого метода в SciPy следующая:

```
import scipy.optimize; import numpy as np
a,b=2,3
```

```
def f(x):     return x**3*np.log(x) - 6*x*np.sin(x)-12
z = scipy.optimize.root_scalar(f, bracket=[a,b], method='bisect')
print("Корень уравнения %.4f" % z.root)
print("Значение функции в точке  %.4f равно %.11f" % (z.root,f(z.root)))
print("Количество итераций %i" % z.iterations)
print("Функция вызывалась %i раз" % z.function_calls)
```

Краткий комментарий. Из обязательных параметров функции root_scalar() здесь только 3: функция, задающая левую часть уравнения $f(x) = 0$, отрезок, на котором отделен корень, и метод решения. Если на концах заданного отрезка функция имеет одинаковые знаки, то генерируется код ошибки.

Результат работы программы:

```
Корень уравнения 2.6846.
Значение функции в точке  2.6846 равно 0.00000000004.
Количество итераций 39.
Функция вызывалась 41 раз.
```

Метод касательных. Следующий метод, заложенный в функцию root_scalar(), – *метод касательных.* В отличие от метода деления отрезка пополам перед применением метода касательных нужно проверить условия сходимости: функция $f(x)$ должна быть дважды непрерывно дифференцируема на отрезке $[a, b]$, на котором отделен корень; и первая, и вторая производные функции $f'(x)$ и $f''(x)$ должны сохранять постоянные знаки на отрезке $[a, b]$. Выполнение этих условий гарантирует, что за определенное число шагов мы уточним корень с любой наперед заданной погрешностью ε. Иногда удается вычислить корень уравнения и в том случае, когда условия сходимости не выполняются.

Рассмотрим алгоритм метода касательных. За x_0 выбирается точка, в которой выполняется условие $f(x_0)f''(x_0) > 0.$ Это либо точка a, либо точка b. Далее вычисляются точки $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ до тех пор, пока не выполнится условие $|x_{n+1} - x_n| < \varepsilon.$ Если это условие выполняется, то x_{n+1} – приближенное значение корня с погрешностью ε.

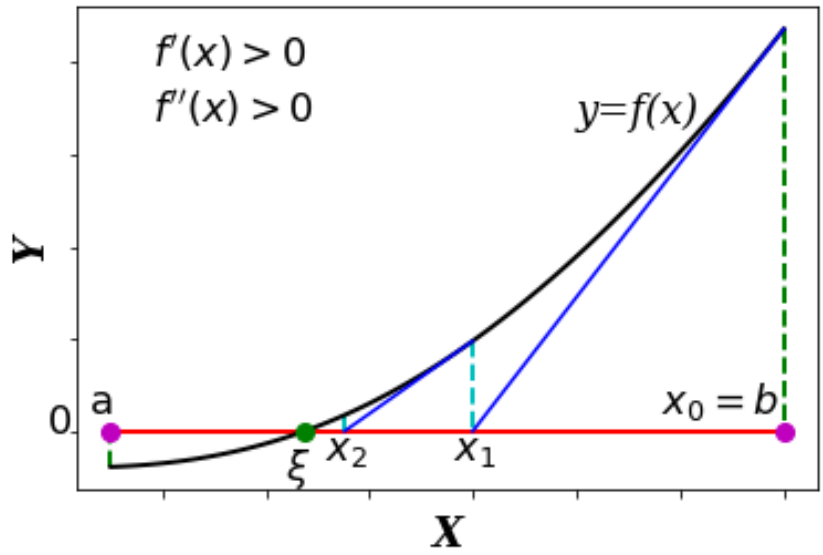

Рис. 2.4. Геометрическая интерпретация метода касательных

Пример 2.2. С погрешностью ε = 0,01 уточнить корень уравнения $x^3 \ln(x) - 6x\sin(x) - 12 = 0$ методом касательных.

Решение. В качестве отрезка, на котором отделен корень уравнения, как и в предыдущем примере, выберем отрезок [2, 3]. Проверим условия сходимости. Для этого построим графики первой и второй производных на этом отрезке. Чтобы не вычислять вручную первую и вторую производные, для построения графика воспользуемся библиотекой SymPy:

```
from sympy import diff, sin,log,plot,symbols
x=symbols("x")
# Построение графика
y=x**3*log(x) - 6*x*sin(x)-12
y1=y.diff(x)
y2=y1.diff(x)
p1=plot(y1,(x,2,3),line_color='r',label="Первая производная",
      xlabel="Ось X", ylabel="Ось Y",
      legend=True,size=(10,7),show=False)
p2=plot(y2,(x,2,3),line_color='b',label="Вторая производная",
      legend=True,show=False)
p1.extend(p2)
p1.show()
```

Результат работы программы приведен на рис. 2.5.

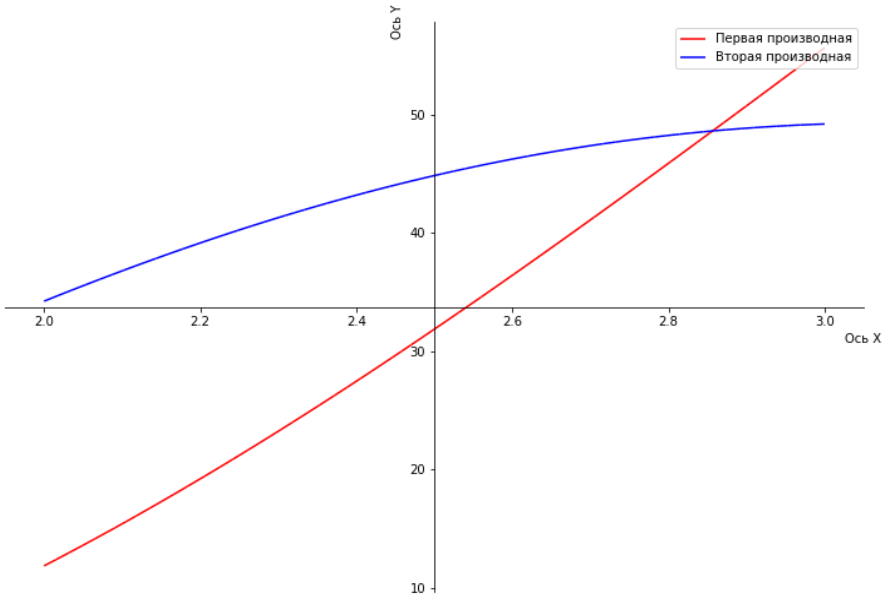

Рис. 2.5. Графики первой и второй производных функции $y=x^3ln(x) - 6xsin(x) - 12$

Комментарий к программе. Импортируем из библиотеки необходимые нам функции. Объявляем переменную *x* символьной. Находим первую (*y1*) и вторую (*y2*) производные от функции *y*. Создаем графические объекты *p1* и *p2* с графиками первой и второй производных. Задаем кривые, графики которых нужно построить, аргумент и пределы его изменения (x,2,3), цвет кривой, текст легенды, подписи по осям *X* и *Y*, графики кривых пока не выводим (show=False). Добавляем второй график к первому (p1.extend(p2)) и выводим оба графика в одно графическое окно. Из графика видно, что обе производные на отрезке положительны, т. е. условия сходимости выполняются.

Выбираем точку x_0 из условия $f(x_0)f''(x_0) > 0$. В нашем случае это точка $b = 3$, так как в этой точке и функция, и вторая производная от нее положительны.

Вычисляем значение x_1 по формуле метода касательных. Для этого понадобится вычислить функцию и производную в точке x_0:

$$f(x) = x^3 \ln(x) - 6x\sin(x) - 12,\ f'(x) = 3x^2 \ln(x) + x^2 - 6\sin(x) - 6x\cos(x).$$

Производную можно получить, выведя на экран значение переменной *y1*:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 3 - \frac{3^3 \ln 3 - 6\cdot 3\sin(3) - 12}{3\cdot 3^2 \ln 3 + 3^2 - 6\sin(3) - 6\cdot 3\cos(3)} = 2.728.$$

$$|x_1 - x_0| = |2.728 - 3| = 0.272 > \varepsilon = 0.01\,.$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 2.728 - \frac{2.728^3\, ln\, 2.728 - 6 \cdot 2.728\, sin(2.728) - 12}{3 \cdot 2.728^2\, ln\, 2.728 + 2.728^2 - 6\, sin(2.728) - 6 \cdot 2.728\, cos(2.728)} = 2.686.$$

$$|x_2 - x_1| = |2.686 - 2.728| = 0.042 > \varepsilon = 0.01.$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)} = 2.686 - \frac{2.686^3\, ln\, 2.686 - 6 \cdot 2.686\, sin(2.686) - 12}{3 \cdot 2.686^2\, ln\, 2.686 + 2.686^2 - 6\, sin(2.686) - 6 \cdot 2.686\, cos(2.686)} = 2.685.$$

$$|x_3 - x_2| = |2.685 - 2.686| = 0.001 < \varepsilon = 0.01.$$

Вывод. Корень уравнения ξ = 2,685 ± 0,001.

Решение уравнения с помощью функции root_scalar():

```
import scipy.optimize
import numpy as np
def f(x):
   return x**3*np.log(x) - 6*x*np.sin(x)-12
def fprime(x):
 return 3*x**2*np.log(x)+x**2-6*np.sin(x)-6*x*np.cos(x)
z = scipy.optimize.root_scalar(f, x0=3, fprime=fprime,
        method='newton')
print("Корень уравнения %.4f" % z.root)
print("Значение функции в точке  %.4f равно %.11f" %
     (z.root,f(z.root)))
print("Количество итераций %i" % z.iterations)
print("Функция вызывалась %i раз" % z.function_calls)
```

Комментарий. При применении этой функции нет необходимости в передаче в функцию отрезка [*a*, *b*], но нужно задать начальное приближение и производную от функции *f*(*x*) (функция fprime).

Результат работы программы:

```
Корень уравнения 2.6846.
Значение функции в точке 2.6846 равно 0.00000000000.
Количество итераций 5.
Функция вызывалась 10 раз.
```

Отметим, что количество итераций значительно сократилось по сравнению с методом деления отрезка пополам.

Возможен и такой вариант использования функции root_scalar():

```
def f(x):
   return x**3*np.log(x) - 6*x*np.sin(x)-12,  \
     3*x**2*np.log(x)+x**2-6*np.sin(x)-6*x*np.cos(x)
z = scipy.optimize.root_scalar(f, x0=3, fprime=True,
        method='newton')
```

В функции $f(x)$ теперь вычисляется и левая часть уравнения, и производная от нее. Это надо учитывать при выводе на экран значения левой части уравнения. Изменилось и значение параметра fprime(): fprime=True. Результаты работы программы не изменились по сравнению с предыдущим вариантом.

Следует отметить, что функция root_scalar() не проверяет ни условия сходимости, ни правильность выбора начального приближения. В нашем примере, если выбрать в качестве x_0 «неправильную» точку $x_0 = 2$, то мы все равно найдем корень уравнения. Однако выбор в качестве начального приближения точки $x_0 = 1.5$ приводит к выводу следующих сообщений:

```
Корень уравнения nan.
Количество итераций 50.
Функция вызывалась 98 раз.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5:
RuntimeWarning: invalid value encountered in log
```

Рассмотрим еще один пример неудачного применения функции root_scalar() для поиска корня уравнения $x^2 + 5 = 0$, не имеющего действительных корней:

```
import scipy.optimize
import numpy as np
def f(x):
   return x**2+5
def fprime(x):  return 2*x
z = scipy.optimize.root_scalar(f, x0=3, fprime=fprime,
        method='newton')
print("Корень уравнения %.4f" % z.root)
print("Значение функции в точке  %.4f равно %.11f" %
    (z.root,f(z.root)))
print("Количество итераций %i" % z.iterations)
print("Функция вызывалась %i раз" % z.function_calls)
```

Сделав максимально возможное количество итераций (или вычислений функции $f(x)$), получаем решение:

```
Корень уравнения 1.1713.
Значение функции в точке 1.1713 равно 6.37184624265.
Количество итераций 50.
Функция вызывалась 100 раз.
```

Очевидно, что полученное значение не является корнем уравнения $x^2 + 5 = 0$, поэтому лучше все-таки перед применением функции, с помощью которой мы хотим найти корень уравнения, убедиться в наличии действительных корней, проверить условия сходимости и правильно задать начальное приближение, а не выбирать его наугад.

Метод Брента. Следующий метод, с помощью которого можно решить уравнение $f(x) = 0$, – *метод Брента*. Для решения уравнения этим методом с помощью функции root_scalar() требуется задание интервала поиска $[a, b]$, на концах которого функция $f(x)$ должна иметь разные знаки. Обычно метод Брента считается лучшей процедурой поиска корня из предложенных. Это гибридный алгоритм поиска корня, сочетающий алгоритмы метода деления отрезка пополам, метода хорд (секущих) и обратную квадратичную интерполяцию. Иногда его называют методом ван Вейнгардена–Деккера–Брента.

Реализация метода Брента в SciPy для примера 2.2:

```
import scipy.optimize; import numpy as np
a,b=2,3
def f(x):     return x**3*np.log(x) - 6*x*np.sin(x)-12
z = scipy.optimize.root_scalar(f, bracket=[a,b], method='brentq')
print("Корень уравнения %.4f" % z.root)
print("Количество итераций %i" % z.iterations)
print("Функция вызывалась %i раз" % z.function_calls)
```

Выполнив программу, получим:

```
Корень уравнения 2.6846.
Количество итераций 8.
Функция вызывалась 9 раз.
```

Метод 'brenth' – модификация метода Брента с гиперболической экстраполяцией. Меняем значение параметра method:

```
z = scipy.optimize.root_scalar(f, bracket=[a,b], method='brenth').
```

Получили тот же результат, что и в методе 'brentq'.

Метод Риддера. В соответствии с методом задача поиска корня уравнения $f(x) = 0$ сводится к решению двух задач: созданию уникальной экспоненциальной функции $g(x)$ вида $g(x) = f(x)e^{(x-x_1)Q}$ и поиску корня уравнения $g(x) = 0$, который будет являться корнем уравнения $f(x) = 0$.

Пример реализации метода:

```
import scipy.optimize; import numpy as np; a,b=2,3
def f(x):
    return x**3*np.log(x) - 6*x*np.sin(x)-12
z = scipy.optimize.root_scalar(f,bracket=[a,b],method='ridder')
print("Корень уравнения %.4f" % z.root)
print("Количество итераций %i" % z.iterations)
print("Функция вызывалась %i раз" % z.function_calls)
```

Выполнив программу, получаем:

```
Корень уравнения 2.6846.
Количество итераций 5.
Функция вызывалась 12 раз.
```

2.2. Функции root() и fsolve() для решения нелинейных уравнений и систем нелинейных уравнений

Для решения уравнений и систем уравнений в SciPy можно использовать функцию root(), имеющую следующий синтаксис:

scipy.optimize.root(fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None, options=None).

Параметры функции root():

fun – функция, задающая левую часть уравнения или левые части системы уравнений, которые необходимо решить;

x0 – массив типа *ndarray*, задающий начальное приближение;

args=() – кортеж аргументов, которые могут входить в функцию, необязательный параметр. Дополнительные аргументы передаются целевой функции и ее якобиану;

method – символьная переменная, задающая метод решения, необязательный параметр. Может принимать значения {'hybr', 'lm', 'broyden1', 'broyden2', 'anderson', 'linearmixing', 'diagbroyden', 'excitingmixing', 'krylov', 'df-sane'};

jac – переменная, имеющая тип bool или callable, необязательный параметр. Если *jac* является логической переменной и принимает значение True, предполагается, что *fun* возвращает значение якобиана вместе со значением целевой функцией. Если значение переменной равно False, якобиан будет оцениваться численно. *jac* также может иметь тип callable и задаваться как отдельная функция, возвращающая значение якобиана функции *fun*. В этом случае он должен принимать те же аргументы, что и *fun*;

tol – желаемая погрешность вычислений. Число типа float, необязательный параметр:

options – словарь, содержащий некоторые дополнительные параметры, например *xtol* или *maxiter*.

Решение выдается в виде объекта OptimizeResult. Важными атрибутами являются: *x* – массив решений, *success* – логическая переменная, указывающая, успешно ли завершился алгоритм, и переменная *message*, описывающая причину завершения.

Рассмотрим пример применения функции root() для решения примера 2.2 с выбором метода решения по умолчанию:

```
import scipy.optimize
import numpy as np
def f(x):
    return x**3*np.log(x) - 6*x*np.sin(x)-12
z = scipy.optimize.root(f, x0=3)
z
```

Выполнив программу, получаем:

```
fjac: array([[-1.]])
     fun: array([3.55271368e-15])
 message: 'The solution converged.'
    nfev: 8
     qtf: array([-2.61953303e-09])
       r: array([-40.36665546])
  status: 1
 success: True
       x: array([2.68461532])
```

Ответ можно получить в более компактном виде, выведя на экран только нужную информацию. Это можно сделать следующим образом:

```
print("Корень уравнения %.4f" % z.x)
print("Значение функции в точке  %.4f равно %.11f" %
    (z.x,z.fun))
print("Количество итераций %i" % z.nfev)
```

Результат:

```
Корень уравнения 2.6846.
Значение функции в точке  2.6846 равно 0.00000000000.
Количество итераций 8.
```

Изменив метод решения с 'hybr' (по умолчанию), к примеру, на 'broyden1' (z = scipy.optimize.root(f, x0=3,method='broyden1')), мы получим ответ в несколько ином виде:

```
 fun: array([4.51135485e-09])
 message: 'A solution was found at the specified tolerance.'
    nit: 8
 status: 1
 success: True
       x: array(2.68461532)
```

Решить уравнение (в том числе и трансцендентное) $f(x) = 0$ в Python можно с помощью функции fsolve(), обязательными параметрами которой являются функция f и начальное приближение x. Если начальное приближение неизвестно, можно, как было показано в подразд. 2.1, построить график функции $y = f(x)$ и определить отрезок, на котором отделен корень уравнения. Функция fsolve() имеет следующий синтаксис:

fsolve(func, x0, args=(), fprime=None, full_output=0, col_deriv=0, xtol=1.49012e-08, maxfev=0, band=None, epsfcn=None, factor=100, diag=None).

Функция возвращает корень (нелинейного) уравнения func(x) = 0 при заданном начальном приближении x0.

Некоторые параметры функции fsolve():

func – функция, которая принимает по крайней мере один (возможно, векторный) аргумент и возвращает значение той же длины;

x0 – переменная типа ndarray. Начальное приближение к корню уравнения func(x) = 0;

args – кортеж, необязательный параметр. С его помощью можно задать дополнительные аргументы для функции *func*;

fprime – функция f(x, *args), необязательный параметр. Функция, вычисляющая якобиан функции *func* с производными по строкам;

full_output – переменная типа bool, необязательный параметр. Если значение переменной равно True, то, кроме решения уравнения, выдается дополнительная информация о решении;

col_deriv – переменная типа bool, необязательный параметр. Используется, если производная в якобиане вычисляется не по строкам, как предполагается по умолчанию, а по столбцам;

xtol – переменная типа float, необязательный параметр. Вычисление корня завершится, если относительная ошибка между двумя последовательными итерациями составит не более *xtol*;

maxfev – переменная типа int. Максимальное количество вызовов функции. Если переменной присвоено значение 0, максимальное значение вычисляется по формуле 100*(N+1), где N – размерность задачи.

Некоторые выходные параметры функции fsolve():

x – переменная типа ndarray. Решение уравнения или результат последней итерации при неудачной попытке решения;

infodict – словарь необязательных выходных параметров с ключами:

fvec – значение функции после последней итерации;

nfev – число вызовов функции;

mesg – переменная типа str. Если решение не найдено, *mesg* подробно описывает причину сбоя.

Покажем, как можно решить пример 2.2 с помощью функции fsolve(), использовав минимальное число входных аргументов. Код для решения задачи:

```
import scipy.optimize; import numpy as np
def f(x):     return x**3*np.log(x) - 6*x*np.sin(x)-12
z = scipy.optimize.fsolve(f, x0=3); z
```

Результатом будет массив с одним элементом – корнем уравнения: (array([2.68461532]).

Если мы хотим получить больше информации о решении, нужно добавить в функцию fsolve() еще один параметр: full_output=True. В результате работы измененной программы получим словарь:

```
 {'fjac': array([[-1.]]),
  'fvec': array([3.55271368e-15]),
  'nfev': 8,
  'qtf': array([-2.61953303e-09]),
```

```
 'r': array([-40.36665546])},
1,
'The solution converged.')
```

Покажем теперь, как можно с помощью функций root() и solve() решить систему нелинейных уравнений.

Пример 2.3. Решить систему нелинейных уравнений $\begin{cases} x^2\sin(x+y)+y^2-2x=4 \\ x^2-y+x=-3\cos(y) \end{cases}$ с помощью функций root() и fsolve().

Решение. Приведем систему к виду $\begin{cases} g_1(x,y)=0 \\ g_2(x,y)=0 \end{cases}$:

$$\begin{cases} x^2\sin(x+y)+y^2-2x-4=0 \\ x^2-y+x+3\cos(y)=0 \end{cases}.$$

Решение систем нелинейных уравнений, как правило, очень чувствительно к выбору начального приближения. Выберем его, построив графики функций $x^2\sin(x+y)+y^2-2x-4=0$ и $x^2-y+x+3\cos(y)=0$. В Python это можно сделать с помощью функции библиотеки SymPy plot_implicit(). Код для построения графика следующий:

```
from sympy import *
x, y = symbols('x y')
p1 = plot_implicit(Eq(x**2*sin(x+y)+y**2-2*x,4),
  (x, -7, 8),adaptive=False,show=False)
p2=plot_implicit(Eq(x**2-y+x,-3*cos(y)),
 (x, -7, 8),adaptive=False,show=False,line_color='red')
p1.append(p2[0]);p1.show()
```

Результат работы программы приведен на рис. 2.6.

Комментарий к программе. Поскольку библиотека SymPy предназначена для работы с символьными переменными, объявляем переменные *x* и *y* символьными. Для создания уравнений с символьными переменными в SymPy используют функцию Eq(). Она содержит два параметра: левую и правую части уравнения. Далее создаются объект p1 с графиком первой кривой и объект p2 с графиком второй кривой. Задаются пределы изменения аргумента *x*: (x, −7, 8). Значения *y* функция plot_implicit() определяет сама из решения соответствующего уравнения. Чтобы задать разный цвет кривых, графики пока не

выводим на экран. После того как в одной из команд plot_implicit() мы изменили цвет графика, добавляем один график к другому и выводим их в одно графическое окно. Параметр adaptive позволяет увеличить толщину линий графика. Из графика видно, что на интервале [−7,8] система уравнений имеет два решения. Найдем одно из них, выбрав в качестве начального приближения вектор [−1, 1].

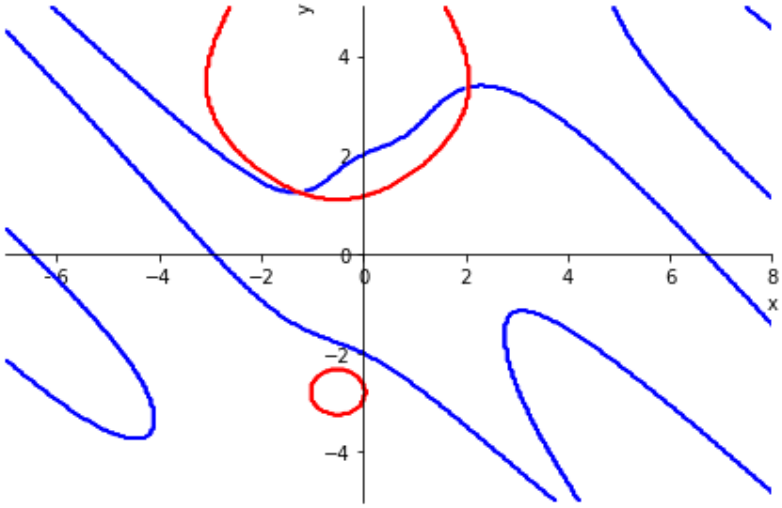

Рис. 2.6. Графики двух неявно заданных функций

В иллюстративных целях приведем решение задачи с помощью функции root() с использованием якобиана, а с помощью функции fsolve() – без его явного задания.

Вычислим якобиан:

$$jac = \begin{pmatrix} \frac{\partial g_1(x,y)}{\partial x} & \frac{\partial g_1(x,y)}{\partial y} \\ \frac{\partial g_2(x,y)}{\partial x} & \frac{\partial g_2(x,y)}{\partial y} \end{pmatrix} = \begin{pmatrix} 2x\sin(x+y)+\cos(x+y)x^2+2 & x^2\cos(x+y)+2y \\ 2x & -1-3\cos(y) \end{pmatrix}.$$

При решении задачи переменную *x* в коде заменяем на x[0], а переменную *y* – на x[1], как этого требует функция root() при решении систем уравнений. Код решения задачи с помощью функции root():

```
from scipy import optimize; import numpy as np
def fun(x):
 return [x[0]**2*np.sin(x[0]+x[1])+ x[1]**2-2*x[0]-4,
         x[0]**2-x[1]+x[0]+3*np.cos(x[1])]
def jac(x):
 return np.array([[2*x[0]*np.sin(x[0]+x[1])+\
```

```
                np.cos(x[0]+x[1])*x[0]**2+2,
        x[0]**2*np.cos(x[0]+x[1])+2*x[1]],
        [2*x[0],
        -1-3*np.cos(x[1])]])
sol = optimize.root(fun, [-1, 1], jac=jac, method='hybr')
print("Решение системы уравнений: x= %.6f y=%.6f" %
     (sol.x[0],sol.x[1]))
print("Значение левой части первого уравнения равно %.12f"
 % sol.fun[0])
print("Значение левой части второго уравнения равно %.12f"
 % sol.fun[1])
print("Количество итераций: %i" % sol.nfev)
```

Решение задачи:

```
Решение системы уравнений: x= -1.215009 y=1.238860.
Значение левой части первого уравнения равно 0.000000000000.
Значение левой части второго уравнения равно -0.000000000000.
Количество итераций: 10.
```

Комментарий к коду. Определяем функцию fun(), в которой задаем левые части каждого из уравнений. С помощью функции jac вычисляем ее якобиан. Решаем задачу методом 'hybr' с начальным приближением [−1,1], найденным из графика (рис. 2.6). Выводим на экран решение системы уравнений, значения левых частей каждого уравнения и количество итераций.

Решим ту же задачу с помощью функции fsolve(). Код программы:

```
import scipy.optimize; import numpy as np
def g(z):
  x=z[0];y=z[1]
  f=np.zeros(2)
  f[0]=x**2*np.sin(x+y)+y**2-2*x-4
  f[1]=x**2-y+x+3*np.cos(y)
  return f
z=scipy.optimize.fsolve(g,[-1,1])
print("Решение системы уравнений: x= %.6f y=%.6f" %
     (z[0],z[1]))
print("Значение левой части первого уравнения равно %.12f "
 % g(z)[0])
```

```
print("Значение левой части второго уравнения равно %.12f "
 % g(z)[1])
```

Выполнив программу, получаем решение системы уравнений:

```
Решение системы уравнений: x= -1.215009 y=1.238860.
Значение левой части первого уравнения равно
-0.000000000007.
Значение левой части второго уравнения равно 0.000000000006.
```

Комментарий к коду. Определяем функцию g(z), в которой задаем левые части каждого из уравнений. Чтобы не писать в уравнении вместо *x* переменную z[0], а вместо *y* – z[1], как того требует синтаксис функции fsolve() в случае решения систем уравнений, оставляем в записи системы уравнений наши переменные *x* и *y*, сделав соответствующие замены. Очищаем значение переменной *f* и присваиваем значения f[0] и f[1] левой части соответственно первого и второго уравнения. Все это оформляем в виде функции, возвращающей значения левых частей системы уравнений. В функции fsolve() задаем два параметра: имя созданной функции g и вектор с начальным приближением. Убеждаемся, что мы получили именно *решение* системы уравнений. Для этого, кроме решения, выводим на экран значения левых частей каждого уравнения.

Задания для самостоятельной работы

Задание 1. Решить алгебраическое уравнение $(\nu + x)^3 - 2x^2 - -\alpha x - 17 = 0$ с использованием функции root(). Получить графическое решение.

Задание 2. Найти корни алгебраического полинома $\alpha x^3 - (\gamma + +\vartheta)x^2 - \mu x + \nu = 0$ с помощью функции fsolve(). Привести графическое решение задачи.

Задание 3. Решить систему линейных уравнений с помощью функции fsolve():

$$\begin{cases} 3x_1 + x_2 + x_3 + \gamma x_4 = \alpha \\ x_1 - \mu x_2 + \nu x_3 + 4x_4 = \beta \\ -5x_1 - x_3 - 7x_4 = -5 \\ x_1 - 6x_2 + \alpha x_3 + 6x_4 = 3 \end{cases}.$$

Задание 4. Решить систему линейных уравнений с помощью функции root():

$$\begin{cases} \beta x_1 + \gamma x_2 + \nu x_3 + x_4 = 10\mu + \nu \\ x_1 - 3\mu x_2 + 2x_3 + 4\alpha x_4 = 9 + \gamma \\ -5x_1 + \nu x_2 - x_3 - 7x_4 = -5 \\ \nu x_1 - 6x_2 + 2x_3 + 6x_4 = \mu \end{cases}.$$

Задание 5. Решить систему нелинейных уравнений. Получить графическое решение:

1. $\begin{cases} 3\ln x + y\sin x + y^2 = 12 \\ \dfrac{x}{y^2+1} + 2^x - 3y^2 = 22 \end{cases}$.

2. $\begin{cases} x^2 - y^3 + \dfrac{x}{\sin y + 2} + 2^x = 15 \\ x\cos^2 y + e^{x-y} - y^2 = 17 \end{cases}$.

3. $\begin{cases} \dfrac{x}{y^2+1} + 2x\cos y + e^{-x} + xy = 38 \\ \dfrac{3xy}{\sin x + 2} + \ln(|y| + 6) - x^2 = 5 \end{cases}$.

4. $\begin{cases} x^2 + y^2 = 5xy + 3 \\ \dfrac{5y}{x^2+1} + 4\cos x = 6 \end{cases}$.

5. $\begin{cases} x^2\cos y = 5xy + 3 \\ \dfrac{5y}{\ln(x^2+1)} + 4x - y = 6 \end{cases}$.

6. $\begin{cases} \sqrt{x}\cos y = 5xy + 3 \\ \dfrac{5y}{x^2 + y^2 + 1} + x - y = 4 \end{cases}$.

3. АППРОКСИМАЦИЯ

Аппроксимация – это замена одной функции $f(x)$ другой, похожей функцией $Y(x)$. Методы аппроксимации применяют как в случае, когда функция $f(x)$ задана в табличном виде, так и тогда, когда функция $f(x)$ является непрерывной и есть необходимость получить упрощенное математическое описание имеющейся сложной зависимости. На практике задача аппроксимации часто возникает тогда, когда по экспериментальным данным требуется подобрать такую аналитическую функцию, которая проходила бы как можно ближе к экспериментальным точкам. Построенная в результате решения задачи аппроксимации кривая сглаживает обрабатываемые экспериментальные данные.

Пусть в результате эксперимента получены данные, представленные в виде табл. 3.1.

Таблица 3.1

Экспериментальные данные

x_i	x_0	x_1	x_2	x_3	…	x_m
y_i	y_0	y_1	y_2	y_3	…	y_m

Необходимо заменить таблично заданную функцию аналитической функцией $Y=f(x,c_0,c_1,\ldots,c_k)$. При этом искомая функция $f(x,c_0,c_1,\ldots,c_k)$ может зависеть от параметров c_j $(j=\overline{0,k})$ как линейно, и тогда говорят о линейной аппроксимации, так и не линейно. В последнем случае говорят о нелинейной аппроксимации. При решении задачи надо таким образом подобрать коэффициенты $c_0, c_1, \ldots, c_k$ функции Y, чтобы отклонения экспериментальных значений y_i от модельных $Y_i = f(x_i,c_0,c_1,\ldots,c_k)$ были в каком-то смысле минимальными. Решение задачи аппроксимации состоит из нескольких этапов. На первом надо определить вид зависимости, т. е. выбрать такую кривую, которая лучше всего подходит для описания экспериментальных данных. При этом исходят или из аналитических предпосылок или выбирают вид кривой визуально исходя из графика с экспериментальными данными. Если выбрать одну из нескольких кривых затруднительно, можно воспользоваться методом средних точек. Метод средних точек заключается в том, что для каждой из кривых, которые в принципе могут быть использованы для решения задачи аппроксимации,

рассчитывается средняя точка. Все эти точки наносятся на график с экспериментальной кривой. Выбирается та кривая, средняя точка которой находится ближе всего к экспериментальной кривой. После того как вид зависимости выбран, необходимо указать способ получения неизвестных коэффициентов c_0, c_1, … , c_k. Число искомых коэффициентов $k+1 \le m$. После их нахождения нужно оценить правильность решения.

Одним из наиболее часто используемых способов аппроксимации кривых для функционально связанных экспериментальных данных является полиномиальная аппроксимация. В этом случае аппроксимирующая функция имеет вид $f(x, c_0, c_1, \ldots, c_k) = c_0 + c_1 x + c_2 x^2 + \cdots + c_k x^k$. Среди других часто используемых способов аппроксимации следует отметить логарифмическую, экспоненциальную, степенную аппроксимацию, аппроксимацию тригонометрическими функциями, в том числе рядами Фурье.

3.1. Меры погрешности аппроксимации

Близость исходной $f(x)$ и аппроксимирующей $Y=f(x,c_0,c_1,\ldots,c_k)$ функций определяется некоторой числовой мерой, называемой критерием аппроксимации или критерием близости. Наиболее часто используемыми подходами к решению задачи аппроксимации являются метод наименьших модулей, минимаксный подход и метод наименьших квадратов.

При решении задачи аппроксимации *методом наименьших модулей* в качестве критерия близости выбирают критерий $M = \sum_{i=0}^{m} \left| y_i - f(x_i, c_0, c_1, \ldots, c_k) \right|$, а коэффициенты c_j ищут из условия $M \to \min\limits_{c_j}$. Минимизируются суммы (иногда – взвешенные) абсолютных значений невязок. Невязка – это разность между экспериментальными и модельными значениями функции $y_i - f(x_i, c_0, c_1, \ldots, c_k)$ $(i = \overline{0, m})$. На рис. 3.1 это отрезки прямых синего цвета.

При решении задачи аппроксимации с использованием *минимаксного подхода* (равномерное приближение) в качестве меры

погрешности выбирают $R = \max_i |y_i - f(x_i, c_0, c_1, ..., c_k)|$. Коэффициенты c_j подбирают таким образом, чтобы R было минимальным: $\min_{c_j} \max_i |y_i - f(x_i, c_0, c_1, ..., c_k)|$. Этот подход осуществляет равномерное приближение функций. Геометрически это означает, что из всех кривых заданного вида выбирают ту, у которой максимальная по модулю невязка минимальна. У кривой, изображенной на рис. 3.2, модуль максимальной невязки равен длине отрезка AB.

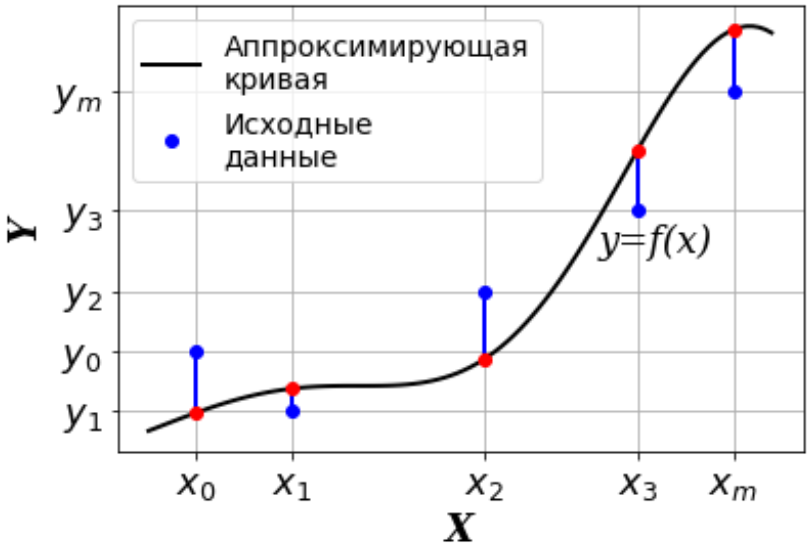

Рис. 3.1. Аппроксимация методом наименьших модулей

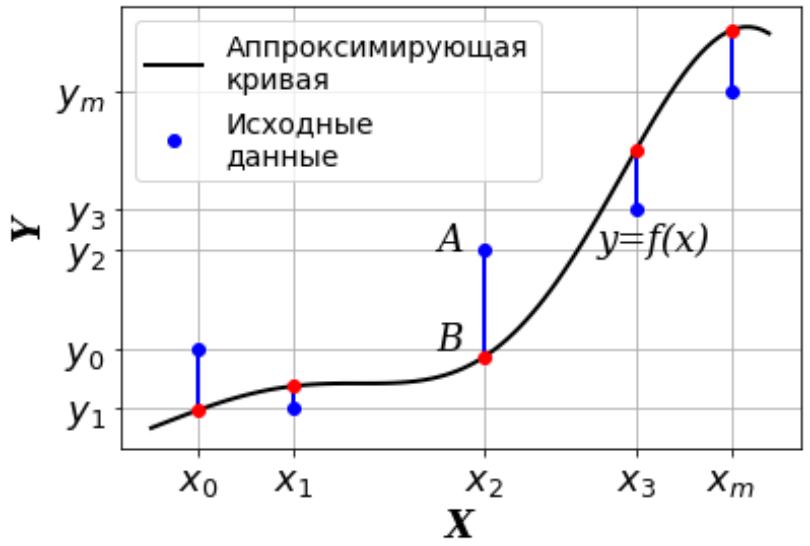

Рис. 3.2. Минимаксный подход к решению задачи аппроксимации

Чаще всего при точечной аппроксимации используют меру $K = \sum_{i=0}^{m} [y_i - f(x_i, c_0, c_1, ..., c_k)]^2$, а коэффициенты c_j $(j = \overline{0,k})$ ищут из условия $K \to \min\limits_{c_j}$. Квадратичный критерий дифференцируем и обеспечивает единственное решение задачи аппроксимации для полиномиальных аппроксимирующих функций. Описанный подход к задаче аппроксимации называется *методом наименьших квадратов*. Геометрически это означает, что из всех кривых заданного вида (полиномов заданной степени, кривых вида $y = ae^{bx}$ и т. д.) выбирают ту, у которой сумма площадей квадратов отклонений наименьшая (рис. 3.3).

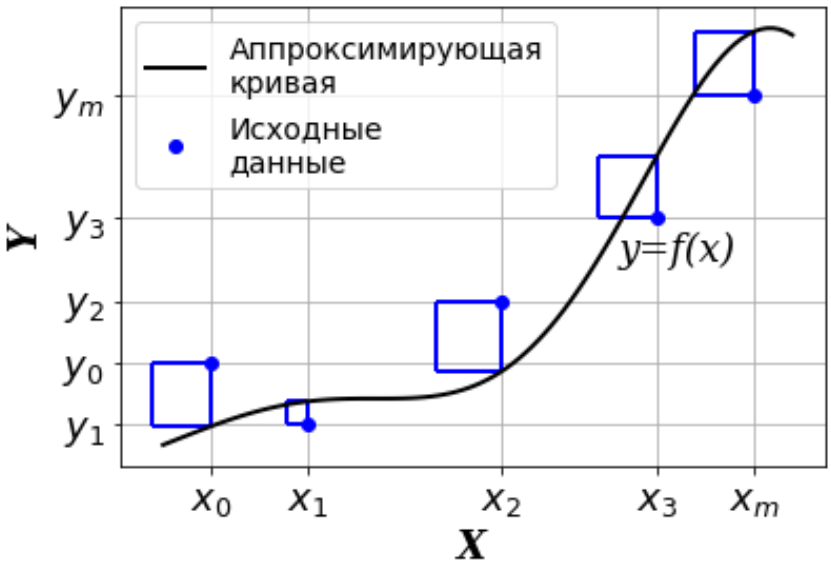

Рис. 3.3. Решение задачи аппроксимации методом наименьших квадратов

Неизвестные коэффициенты c_j $(j = \overline{0,k})$ можно найти, приравняв к нулю частные производные по этим коэффициентам: $\frac{\partial K}{\partial c_j} = 0,\ j = \overline{0,k}.$

В данном пособии остановимся лишь на решении задачи аппроксимации методом наименьших квадратов. При этом всюду будем предполагать, что вид аппроксимирующей кривой нам известен.

Решение задачи аппроксимации с использованием других подходов можно найти в учебниках по численным методам[1].

[1] Самарский А. А., Гулин А. В. Численные методы. Москва: Наука, 1989. 432 с.; Мудров В. И., Кушко В. Л. Метод наименьших модулей. Москва: Знание, 1971. 64 с.

3.2. Метод наименьших квадратов для решения задачи аппроксимации. Функции linalg.lstsq(), curve_fit() и optimize.leastsq()

Сначала рассмотрим решение задачи точечной аппроксимации методом наименьших квадратов в случае, когда апппроксимирующая кривая является алгебраическим полиномом. Это, как правило, самый простой способ аппроксимации, когда решение можно найти и без применения специальных функций. Покажем, как это можно сделать, на примере.

Пример 3.1. Методом наименьших квадратов построить аппроксимирующую параболу $y^m=c_0+c_1x+c_2x^2$ для функции, заданной в табличном виде:

x	0	1	2	3
y	−1,2	−0,1	1,3	1,8

Найти сумму площадей квадратов отклонений. Решить задачу с помощью функции lstsq() модуля scipy.linalg. Дать графическое решение задачи.

Решение. Поскольку задано 4 узла таблицы, можно заменять такую таблично заданную функцию полиномами до 3-й степени. В соответствии с условием задачи нужно подобрать такой полином 2-й степени $y^m=c_0+c_1x+c_2x^2$, чтобы $K=\sum_{i=0}^{m}\left(y_i-c_0-c_1x_i-c_2x_i^2\right)^2\to\min\limits_{c_0,c_1,c_2}$, где x_i, y_i – заданные точки таблицы. Неизвестные коэффициенты найдем из решения системы нормальных уравнений, получить которую можно, приравняв к нулю частные производные от K по неизвестным коэффициентам:

$$\begin{cases}\dfrac{\partial K}{\partial c_0}=-2\sum\limits_{i=0}^{3}\left(y_i-c_0-c_1x_i-c_2x_i^2\right)=0\\ \dfrac{\partial K}{\partial c_1}=-2\sum\limits_{i=0}^{3}\left(y_i-c_0-c_1x_i-c_2x_i^2\right)x_i=0\\ \dfrac{\partial K}{\partial c_2}=-2\sum\limits_{i=0}^{3}\left(y_i-c_0-c_1x_i-c_2x_i^2\right)x_i^2=0\end{cases}.$$

После преобразований получаем

$$\begin{cases} \sum_{i=0}^{3} y_i = (3+1)c_0 + c_1\sum_{i=0}^{3} x_i + c_2\sum_{i=0}^{3} x_i^2 \\ \sum_{i=0}^{3} x_i y_i = c_0\sum_{i=0}^{3} x_i + c_1\sum_{i=0}^{3} x_i^2 + c_2\sum_{i=0}^{3} x_i^3 \\ \sum_{i=0}^{3} x_i^2 y_i = c_0\sum_{i=0}^{3} x_i^2 + c_1\sum_{i=0}^{3} x_i^3 + c_2\sum_{i=0}^{3} x_i^4 \end{cases}.$$

Для нахождения сумм составим таблицу

x	y	xy	x^2y	x^2	x^3	x^4
0	-1,2	0	0	0	0	0
1	-0,1	-0,1	-0,1	1	1	1
2	1,3	2,6	5,2	4	8	16
3	1,8	5,4	16,2	9	27	81
6	**1,8**	**7,9**	**21,3**	**14**	**36**	**98**

Подставив полученные значения в систему уравнений, получаем

$$\begin{cases} 1{,}8 = 4c_0 + 6c_1 + 14c_2 \\ 7{,}9 = 6c_0 + 14c_1 + 36c_2 \\ 21{,}3 = 14c_0 + 36c_1 + 98c_2 \end{cases}.$$

Решаем эту систему уравнений, например, с помощью функции solve() модуля scipy.linalg:

```
import numpy as np;from scipy.linalg import *
A = np.array([ [4,6,14], [6,14,36], [14,36,98]])
# Преобразуем строку в столбец
b = np.array([1.8,7.9,21.3]).reshape(3,1)
c=solve(A,b)
for i in range(len(b)):
  print("c(%i)=%.7f" % (i,c[i]))
```

Получаем решение:

```
c(0)=-1.2600000
c(1)=1.4900000
c(2)=-0.1500000
```

Таким образом, уравнение аппроксимирующей параболы $y^m = -1.26+1.49x-0.15x^2$. Вычислив значения этой функции при

x = [0, 1, 2, 3], получаем y=[-1.26, 0.08, 1.12, 1.86]. Сумма квадратов невязок равна $(-1.26+1.2)^2 + (0.08+0.1)^2 + (1.12-1.3)^2 + (1.86-1.8)^2 = 0.072$. У всех других парабол 2-й степени сумма площадей квадратов отклонений будет больше, чем 0.072.

Решим теперь задачу аппроксимации в модуле scipy.linalg. Используем для этого метод lstsq(). Он позволяет найти такой вектор c, который минимизирует невязку системы линейных алгебраических уравнений (СЛАУ) $Ac=b$ методом наименьших квадратов (МНК).

Составим матрицу A и вектор b следующим образом:

$$A = \begin{pmatrix} 1\, x_0\ x_0^2 \\ 1\, x_1\ x_1^2 \\ 1\, x_2\ x_2^2 \\ \ldots\ldots\ldots \\ 1\, x_m\ x_m^2 \end{pmatrix}, b = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \ldots \\ y_m \end{pmatrix}.$$

Запишем систему линейных уравнений в матричной форме $Ac=b$, где $c = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$ – вектор-столбец неизвестных коэффициентов. Умножение A на c дает $y_i^m = c_0 + c_1 x_i + c_2 x_i^2, (i = \overline{0,m})$ – значения аппроксимирующей параболы в заданных точках таблицы. Правая часть уравнения – табличные значения функции y_i. Разница между ними дает вектор невязок.

Полученную систему решаем с помощью функции lstsq(). Синтаксис функции lstsq():

lstsq(a, b, rcond='warn').

Обязательными входными параметрами являются матрица A указанного выше вида и вектор значений функции y. Результатом решения являются вектор c со значениями искомых коэффициентов, сумма площадей квадратов отклонений, ранг матрицы A и сингулярные числа матрицы A. Сингулярными числами вещественной матрицы A размерности m на n называются арифметические значения квадратных корней из собственных чисел $\lambda_1, \lambda_2, \ldots, \lambda_k$ матрицы A^TA, где k=min(n,m).

Код программы решения задачи:

```
import scipy.linalg
import numpy as np
```

```
from numpy import *
import matplotlib.pyplot as plt
fig = plt.figure();ax = fig.add_subplot(111)
x=np.array([0, 1, 2, 3])
y=np.array([-1.2, -0.1, 1.3, 1.8])
A=x[:,np.newaxis]**[0,1,2]
res=linalg.lstsq(A,y,rcond=None)
print("Значения коэффициентов")
for i in range(3):
  print("c(%i)=%.7f" % (i,res[0][i]))
a, b, c = res[0][0], res[0][1], res[0][2]
print("Уравнение кривой Ym=%.6f+(%.6f)*x+(%.6f)*x^2" %
(a,b,c))
v=res[1]
print("Сумма площадей квадратов отклонений равна  %.6f" % v)
x1 = np.linspace(min(x), max(x), 100);
ym= a+b*x1+c*x1**2
plt.plot(x, y, 'bo', label="Исходные\nданные")
plt.plot(x1,ym,  label="Аппроксимирующая\nкривая",c='g')
plt.legend()
plt.show()
```

Результатами решения являются значения коэффициентов, уравнение аппроксимирующей параболы, сумма площадей квадратов отклонений и график с исходными данными и аппроксимирующей параболой.

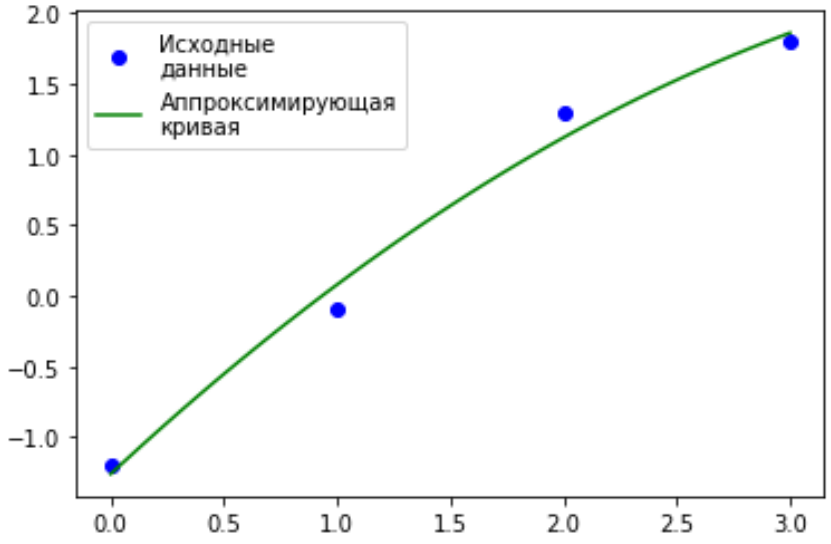

Рис. 3.4. Графическое решение задачи аппроксимации

Значения коэффициентов:

c(0)=-1.2600000
c(1)=1.4900000
c(2)=-0.1500000
Уравнение кривой: Ym=-1.260000+(1.490000)*x+(-0.150000)*x^2.
Сумма площадей квадратов отклонений равна 0.072000.

Комментарий к программе. Подгружаем необходимые библиотеки и модули. Задаем исходные данные (*x* и *y*). В коде мы использовали объект библиотеки Numpy newaxis, с помощью которого можно создать новые оси. Применение его дало нам искомую матрицу *A*:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix}.$$

Решаем задачу аппроксимации с помощью функции linalg.lstsq(), параметрами которой являются созданная матрица *A* и вектор *y*. Параметру rcond присваиваем значение None, чтобы не получать ненужных нам в этой задаче сообщений. Результатами решения являются вектор res[0], элементы которого – искомые коэффициенты и res[1], содержащий значение суммы площадей квадратов отклонений. Другие компоненты решения в данной задаче нам не требуются. Далее выводим полученные результаты и строим график с исходными данными и аппроксимирующей параболой.

Таким способом можно искать решение задачи аппроксимации в виде линейной комбинации любых функций, т. е. в случае, когда аппроксимирующая кривая выглядит так: $Y^m(x) = g(x) = c_0 g_0(x) + c_1 g_1(x) + \ldots + c_k g_k(x)$. Соответственно, матрица *A* будет иметь вид

$$A = \begin{pmatrix} g_0(x_0) & \ldots & g_k(x_0) \\ g_0(x_1) & \ldots & g_k(x_1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ g_0(x_m) & \ldots & g_k(x_m) \end{pmatrix}.$$

Для поиска коэффициентов выбранной зависимости методом наименьших квадратов в SciPy есть еще одна функция – curve_fit(). Ее можно использовать как в случае, когда аппроксимирующая кривая является алгебраическим полиномом, так и в случае неполиномиальной регрессии.

Синтаксис функции curve_fit():

curve_fit(f,xdata,ydata,p0=None,sigma=None,absolute_sigma = False, check_finite=True, bounds=(-np.inf,np.inf),method=None, jac=None, **kwargs).

Основными входными параметрами функции являются:

f – имя аппроксимирующей функции. Функция должна содержать независимую переменную в качестве первого аргумента. Искомые параметры должны быть остальными параметрами функции *f*;

xdata – независимая переменная, массив или последовательность исходных данных;

ydata – зависимые данные, массив или последовательность исходных данных. Значения функции;

p0 – массив или последовательность. Начальное приближение. Если установлено значение None или параметр опущен, то все начальные значения будут равны 1.

Выходными параметрами функции curve_fit() являются массив *popt*, содержащий оптимальные в смысле метода наименьших квадратов значения искомых коэффициентов выбранной зависимости, и *pcov* – ковариационная матрица.

Решим с помощью данной функции задачу нелинейной аппроксимации.

Пример 3.2. Методом наименьших квадратов определить коэффициенты аппроксимирующей кривой $Y=c_1x*cos(c_2x)+c_3$ для таблично заданной функции:

x	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5	5.5	6
y	5.02	6.08	3.33	−0.93	−0.22	7.83	16.52	15.55	2.67	−11.42	−11.78	5.09	25.25

Построить график полученной кривой, на который необходимо нанести исходные данные. Вычислить сумму площадей квадратных отклонений.

Решение. Код программы:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
from scipy.optimize import *
x=np.linspace(0,6,13);
y=np.array([5.02, 6.08, 3.33, -0.93, -0.22, 7.83,
```

```
    16.52, 15.55, 2.67, -11.42, -11.78, 5.09, 25.25])
def f(x, a, b, c):
    return  a*x*np.cos(b*x)+c;
args,_  = curve_fit(f, x, y)
a, b, c = args[0], args[1], args[2]
print("Коэффициенты кривой:")
print("a= %.6f" % a)
print("b= %.6f" % b)
print("c= %.6f" % c)
print("Уравнение кривой Y=%.6f*x*cos(%.6fx)+%.6f" % (a,b,c))
v=np.sum((f(x, a, b, c)-y)**2)
print("Сумма площадей квадратов отклонений равна  %.6f" % v)
plt.plot(x, y, 'bo', label="Исходные\nданные")
plt.plot(x, f(x, a, b, c), label="Аппроксимирующая\nкривая")
plt.legend()
plt.show()
```

Выполнив программу, получаем коэффициенты кривой, ее уравнение и график:

Коэффициенты кривой:

a= 1.802327
b= 0.833727
c= 6.899873
Уравнение кривой: Y=1.802327*x*cos(0.833727x)+6.899873.
Сумма площадей квадратов отклонений равна 1148.003767.

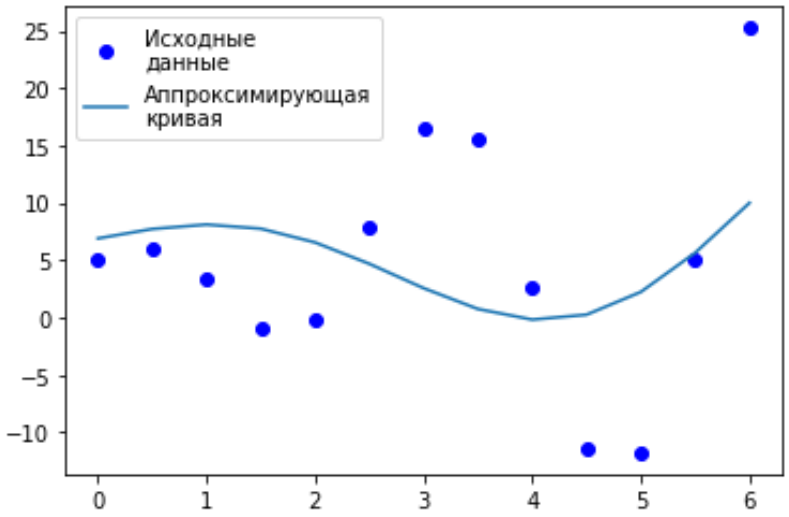

Рис. 3.5. Решение задачи аппроксимации с автоматическим выбором начального приближения

Из рис. 3.5 видно, что полученное решение неудовлетворительное, о чем говорит и сумма площадей квадратов отклонений. Улучшить решение поможет «правильное» задание начального приближения. Таким приближением может быть следующее: a = 1, b = 2, c = 3. Для наглядности увеличим и количество точек, по которым будем строить график аппроксимирующей кривой. Изменения были внесены в следующие операторы:

```
args,_  = curve_fit(f, x, y, [1,2,3])
x1=np.linspace(0,6,100);
plt.plot(x1, f(x1, a, b, c),
        label="Аппроксимирующая\nкривая",c='g')
```

Результат графического решения представлен на рис. 3.6.

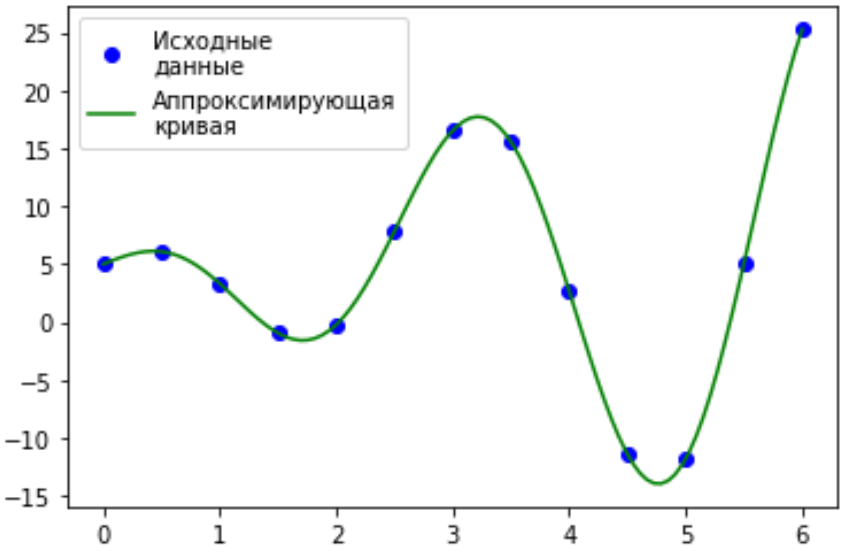

Рис. 3.6. Решение задачи аппроксимации с начальным приближением p0=[1, 2, 3]

Коэффициенты кривой:

a= 4.000196
b= 1.999998
c= 4.999244
Уравнение кривой: Y=4.000196*x*cos(1.999998x)+4.999244.
Сумма площадей квадратов отклонений равна 0.001124.

Анализ результатов решения показывает, что найдено вполне приемлемое решение.

Вывод. Мы вновь убедились, как важно при решении задач численными методами правильно подобрать начальное приближение.

Пример 3.2 можно решить и с помощью функции библиотеки scipy optimize.leastsq(), которая находит минимум квадратов невязок одним из численных методов. Для ее использования нужно сформировать функцию, вычисляющую невязки, и передать ее в качестве первого параметра функции optimize.leastsq(). Второй параметр функции optimize.leastsq() – вектор начальных приближений. В коде это кортеж x0_init.

Программа нахождения необходимых значений и построения соответствующих графиков:

```
from scipy import optimize
import matplotlib.pyplot as plt
import numpy as np
# задаем исходные данные x и y
x=np.linspace(0,6,13);
y=[5.02,6.08,3.33,-0.93,-0.22,7.83,16.52,15.55,
   2.67,-11.42,-11.78,5.09,25.25];
 # задаем аппроксимирующую кривую как функцию
# аргумента x и трех искомых параметров a, b и c
def f(x, a, b, c):
    return a*x*np.cos(b*x)+c;
def g(x0):
    return y- f(x, *x0) # определяем функцию невязок
x0_init = (1,2,3) # задаем начальное приближение
# передаем в функцию leastsq требуемые параметры
x_opt, _ = optimize.leastsq(g, x0_init)
a, b, c =x_opt[0], x_opt[1], x_opt[2]
print("Коэффициенты подобранной зависимости:")
print("a= %.6f" % a)
print("b= %.6f" % b)
print("c= %.6f" % c)
print("Уравнение кривой Y=%.6f*x*cos(%.6fx)+%.6f" % (a,b,c))
v=np.sum((f(x, a, b, c)-y)**2)
print("Сумма площадей квадратов отклонений равна  %.6f" % v)
x1=np.linspace(0,6,40) # задаем пределы значений x
# для построения графика полученной кривой
w=f(x1,a,b,c) # вычисляем
# значения полученной кривой в этих точках и строим график
```

```
plt.plot(x1,w,'r')
plt.plot(x,y,'o')
plt.title("График аппроксимирующей кривой") # заголовок
plt.xlabel("x") # ось абсцисс
plt.ylabel("y") # ось ординат
plt.grid() # наносим на график сетку
plt.legend(('Аппроксимирующая кривая', 'Исходные данные'))
plt.show()
```

Результат работы программы:
Коэффициенты подобранной зависимости:

a= 4.000196
b= 1.999998
c= 4.999244
Уравнение кривой: Y=4.000196*x*cos(1.999998x)+4.999244.
Сумма площадей квадратов отклонений равна 0.001124.

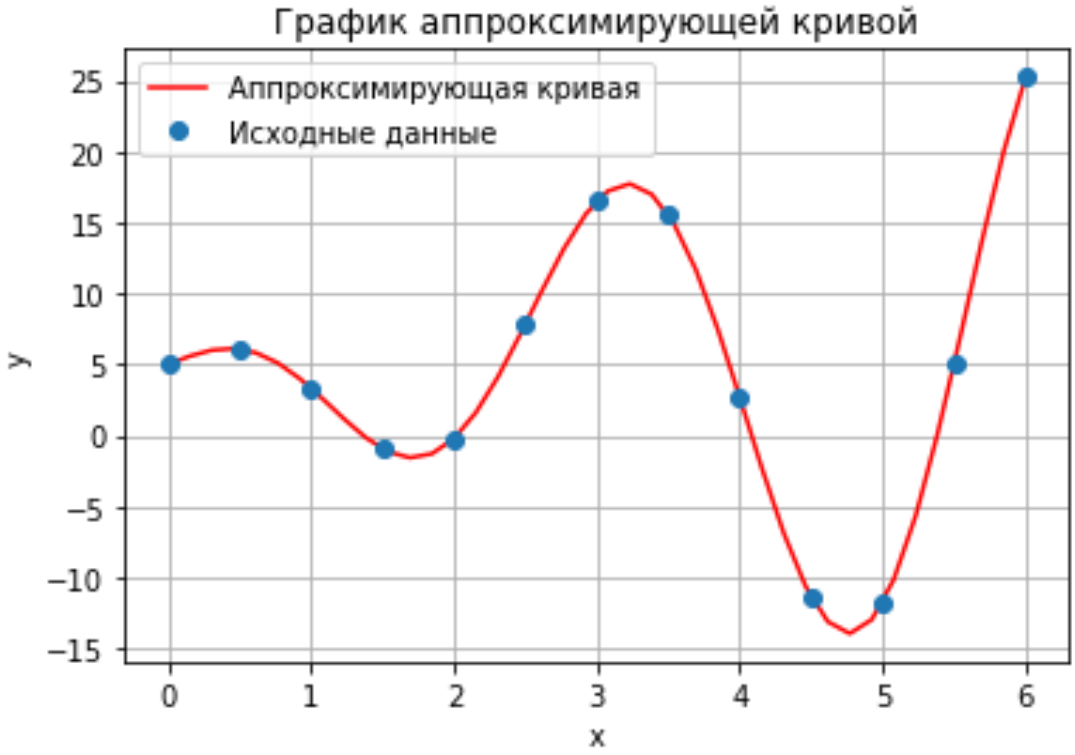

*Рис. 3.7. График аппроксимирующей кривой y=4.0002x·cos(2*x)+4.9992*

Еще раз отметим, что полученное решение существенно зависит от того, насколько хорошо подобрано начальное приближение.

Задания для самостоятельной работы

Задание 1. Методом наименьших квадратов (МНК) получить формулу аппроксимирующей параболы $y = c_0 + c_1 x + c_2 x^2$ для следующей таблицы:

x	μ+ν	μ+ν+2	μ+ν+4	μ+ν+6	μ+ν+8
y	γ+θ+2.5	γ+θ+4.9	γ+θ+8	γ+θ+12.1	γ+θ+16.9

Параметры кривой подобрать методом наименьших квадратов, составив и решив систему нормальных уравнений. Задачу решить также с помощью функции linalg.lstsq(). Вычислить сумму квадратов отклонений между экспериментальными и модельными данными. Для тех же данных построить аппроксимирующую кривую вида $y = ab^x$. Параметры кривой также подобрать методом наименьших квадратов. Нанести на график обе полученные кривые и исходные данные. Сравнить полученные результаты.

Задание 2. Методом наименьших квадратов (МНК) получить формулу аппроксимирующей кривой $y = ax^b$ для следующей таблицы:

x	μ+ν	μ+ν+2	μ+ν+4	μ+ν+6
y	γ+θ+2.5	γ+θ+4.9	γ+θ+8	γ+θ+12.1

Задачу решить с помощью функций optimize.leastsq() и curve_fit(). Нанести полученную кривую и исходные данные на график.

Задание 3. Методом наименьших квадратов получить уравнение аппроксимирующей параболы $y = c_0 + c_1 x + c_2 x^2$, экспоненциальной кривой $y = ae^{bx}$, логарифмической кривой $y = a + b\ln x$ для следующей таблицы:

x	μ+ν	μ+ν+1	μ+ν+2	μ+ν+3	μ+ν+4
y	γ+β	γ+β+1	γ+β+3	γ+β+4	γ+β+8

Для нахождения коэффициентов указанных кривых применить одну из рассмотренных в гл. 3 функций библиотеки SciPy. В каждом случае найти значение критерия *K*. Построить графики получившихся моделей, добавив на каждый график уравнения кривых.

4. ИНТЕРПОЛЯЦИЯ

Интерполяция – частный случай аппроксимации, когда аппроксимирующая кривая проходит через все точки таблицы:

$$f(x, c_0, c_1, \ldots, c_n) = f(x_i), i = \overline{0, n}.$$

Задача интерполяции ставится следующим образом: функция $y = f(x)$ задана в виде таблицы

x	x_0	x_1	…	x_n
y	y_0	y_1	…	y_n

Требуется вычислить значение функции в точке x^*, не совпадающей ни с одним из узлов таблицы. Если эта точка лежит между узлами, то такая задача называется интерполяцией. Если точка x^* лежит за пределами таблицы, то говорят о задаче экстраполяции. В дальнейшем будем предполагать, что значения аргумента x расположены в строго возрастающем порядке: $x_0 < x_1 < x_2 < \ldots < x_n$.

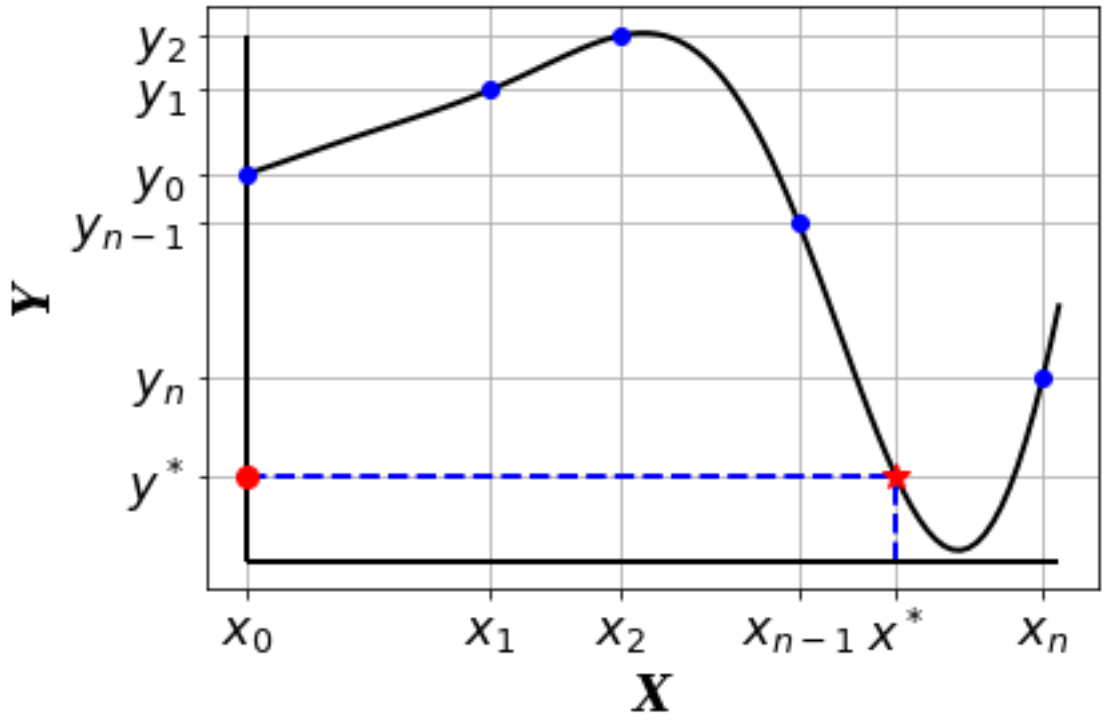

Рис. 4.1. Графическое решение задачи интерполяции

В общем случае в качестве аппроксимирующей кривой могут быть выбраны полиномы, в том числе тригонометрические, рациональные функции и т. д.

4.1. Методы решения задачи интерполяции

Покажем, как можно построить интерполяционную кривую, не прибегая к специальным функциям библиотеки SciPy, предназначенным для решения задачи интерполяции.

Пример 4.1. Вычислить значение функции в точке $x^* = 2.7$. При решении задачи интерполяции аппроксимирующую кривую задать в виде $Y(x) = ax^2 + bx + ce^{dx}$. Исходные данные взять из следующей таблицы:

x	1	2	3	4
y	2	7	25	78

Решение. Решение задачи интерполяции состоит из двух этапов. На первом из них строится кривая, проходящая через все точки таблицы. Вид кривой задан. Необходимо найти неизвестные коэффициенты *a, b, c* и *d.* Эту задачу можно свести к составлению и решению системы нелинейных (в данной случае) уравнений:

$$\begin{cases} Y(x_0) = y_0 \\ Y(x_1) = y_1 \\ Y(x_2) = y_2 \\ Y(x_3) = y_3 \end{cases},$$

где (x_i, y_i) – заданные точки таблицы. Подставляя в систему исходные данные, получаем

$$\begin{cases} a + b + ce^{d} = 2 \\ 4a + 2b + ce^{2d} = 7 \\ 9a + 3b + ce^{3d} = 25 \\ 16a + 4b + ce^{4d} = 78 \end{cases}.$$

Решение этой системы дает следующие результаты: a= -0.081791, b=-2.630986, c=1.764241, d=0.982557. Таким образом, уравнение интерполяционной кривой: $Y(x) = -0.081791x^2 - 2.630986x + 1.764241e^{0.982557x}$. На втором этапе подставляем в это уравнение значение x^*=2.7, получаем $Y(2.7) = 17.343825$.

Рассмотрим код решения этой задачи в Python. Решение системы уравнений найдено с помощью функции fsolve() модуля SciPy optimize. Получено графическое решение задачи:

```
import scipy.optimize
import numpy as np
import matplotlib.pyplot as plt
x=np.array([1,2,3,4]);y=np.array([2,7,25,78])
def w(t,a,b,c,d):
   return a*t**2+b*t+c*np.exp(d*t)
def g(z):
  a=z[0]
  b=z[1]
  c=z[2]
  d=z[3]
  f=np.zeros(4)
  for i in range(len(x)):
    f[i]=w(x[i],a,b,c,d)-y[i]
  return f
z=scipy.optimize.fsolve(g,[1,1,1,1])
print("Решение системы уравнений: a= %.6f b=%.6f c=%.6f
d=%.6f" %
      (z[0],z[1],z[2],z[3]))
print("Значение левой части первого уравнения равно %.12f "
 % g(z)[0])
print("Значение левой части второго уравнения равно %.12f "
 % g(z)[1])
print("Значение левой части третьего уравнения равно %.12f "
 % g(z)[2])
print("Значение левой части четвертого уравнения равно %.10f "
 % g(z)[3])
print("Уравнение кривой Y=(%.6f)x^2+(%.6f)x+%.6f*exp(%.6fx)"
 % (z[0],z[1],z[2],z[3]))
p=2.7;q=w(p,z[0],z[1],z[2],z[3]);
```

```
print("Значение функции в точке %.1f равно %.6f" % (p,q))
fig = plt.figure();ax = fig.add_subplot(111)
a1,b1=min(x), max(x)
x1 = np.linspace(a1, b1, 100);
plt.grid()      # включение отображение сетки
plt.plot(x,y,'ob',label="Исходные\nданные")
plt.plot(x1, w(x1,z[0],z[1],z[2],z[3]),linewidth=2,
        c='k',label="Аппроксимирующая\nкривая")
plt.plot(p, q,"*r",ms=10,label="Решение")
plt.legend()
plt.show()
```

Результаты решения:

Решение системы уравнений: a=-0.081791 b=-2.630986 c=1.764241 d=0.982557.

Значение левой части первого уравнения равно 0.000000000010.

Значение левой части второго уравнения равно 0.000000000055.

Значение левой части третьего уравнения равно 0.000000000233.

Значение левой части четвертого уравнения равно 0.0000000009.

Уравнение кривой: Y=(-0.081791)x^2+(-2.630986)x+ +1.764241*exp(0.982557x).

Значение функции в точке 2.7 равно 17.343825.

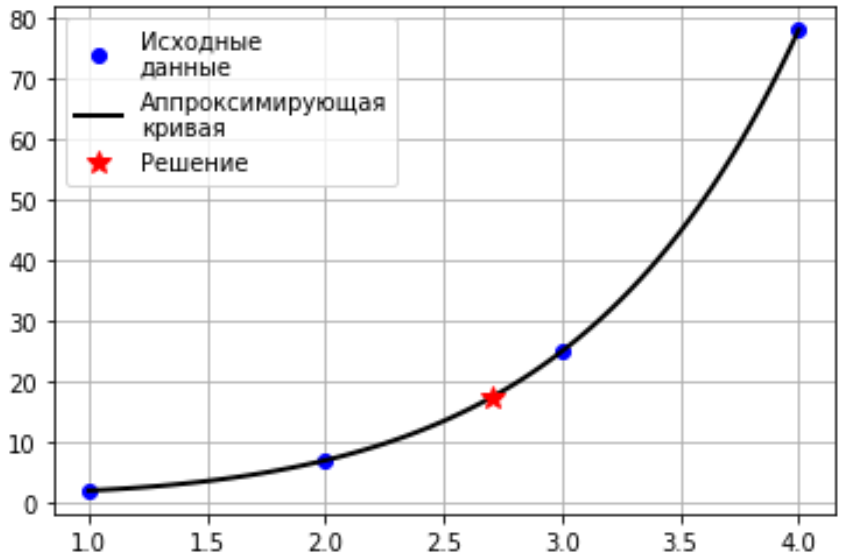

Рис. 4.2. Решение задачи интерполяции

Очевидно, что при увеличении числа неизвестных коэффициентов функции $f(x,c_0,c_1,\ldots,c_n)$, используемой для решения задачи

интерполяции, сложность решения значительно возрастает. Кроме того, как уже отмечалось в гл. 2, правильность решения получившейся системы уравнений во многом зависит от удачно подобранного начального приближения.

Рассмотрим случай, когда аппроксимирующая кривая – алгебраический полином: $Y(x,c)=c_0+c_1x+c_2x^2+\cdots+c_nx^n$. При сделанном предположении (аппроксимирующая функция – полином) очевидно, что коэффициенты c_i можно найти методом наименьших квадратов, положив $n=m$. Это приведет к решению системы нормальных уравнений. Другой способ решении задачи тоже приводит к решению системы уравнений $Y(x_i,c)=y_i,\ i=\overline{0,n}$.

Так мы решали пример 4.1. Существуют и иные способы построения интерполяционных полиномов, не требующие решения систем уравнений.

Запишем уравнение алгебраическиго полинома в виде $L_n(x)=\sum_{i=0}^{n} y_i \prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$. Это выражение – полином Лагранжа степени n. Предположим, что функция задана в виде таблицы из трех точек (x_i, y_i), i = 0,1,2. Запишем уравнение алгебраического полинома 2-й степени:

$$L_2(x)=y_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}+y_1\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}+y_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Нетрудно убедиться, что он является интерполяционным полиномом, т. е. проходит через все точки таблицы: $L_2(x_0)=y_0,\ L_2(x_1)=y_1,\ L_2(x_2)=y_2$. Для того чтобы найти значение функции в некоторой точке x^*, которой нет в таблице, достаточно подставить эту точку вместо x в формулу интерполяционного полинома и считать, что $y^*=L_n(x^*)$. Хотя интерполяционный полином Лагранжа и проходит через заданные точки таблицы, в остальных точках отрезка [x_0, x_n] он отличается от значений функции $g(x)$, которую он заменяет, на величину остаточного члена $R_n(x)$, который определяет абсолютную погрешность формулы Лагранжа:

$$\left|R_n(x)\right|\le\frac{\max\limits_{x_0\le x\le x_n}\left|g^{(n+1)}(x)\right|}{(n+1)!}\left|\prod_{i=0}^{n}(x-x_i)\right|,$$

где $g^{(n+1)}(x)$ – $(n+1)$ – производная функции $g(x)$.

Покажем, как решается задача интерполяции с помощью интерполяционного полинома Лагранжа (ИПЛ) на примере.

Пример 4.2. Заданы значения функции $g(x) = ln(x)$ в трех точках:

x	2	3	4
y	0.69315	1.09861	1.38629

Вычислить значение функции в точке $x^* = 2.5$ с помощью интерполяционного полинома Лагранжа.

Решение. Степень интерполяционного полинома на единицу меньше числа узлов таблицы, поэтому будем строить ИПЛ 2-й степени. Запишем формулу ИПЛ:

$$L_2(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Подставляя в формулу данные таблицы, получаем

$$L_2(x) = 0.69315\frac{(x-3)(x-4)}{(2-3)(2-4)} + 1.09861\frac{(x-2)(x-4)}{(3-2)(3-4)} + 1.38629\frac{(x-2)(x-3)}{(4-2)(4-3)} =$$

$$= \frac{0.69315}{2}(x^2 - 7x + 12) - 1.09861(x^2 - 6x + 8) + \frac{1.38629}{2}(x^2 - 5x + 6) =$$

$$= -0.05889x^2 + 0.69991x - 0.47111.$$

Подставляем в получившуюся формулу значение $x^* = 2.5$:

$$L_2(2.5) = -0.05889 \cdot 2.5^2 + 0.69991 \cdot 2.5 - 0.47111 = 0.9106025.$$

Если шаг в таблице постоянный, то для решения задачи интерполяции можно воспользоваться одной из формул интерполяционного полинома Ньютона (ИПН). Для этого нам понадобится понятие «конечная разность». Конечной разностью функции $y = f(x)$ с шагом $\Delta x = h$ называют функцию $\Delta y = f(x+h) - f(x)$. Это первая конечная разность. Вторая конечная разность

$$\Delta^2 y = [f(x+2h) - f(x+h)] - [f(x+h) - f(x)] = f(x+2h) - 2f(x+h) + f(x)$$ и т. д.

Предположим, что функция $y = f(x)$ задана в виде таблицы из 4 точек. Построим для нее таблицу конечных разностей:

i	x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$
0	x_0	y_0	Δy_0	$\Delta^2 y_0$	$\Delta^3 y_0$
1	x_1	y_1	Δy_1	$\Delta^2 y_1$	
2	x_2	y_2	Δy_2		
3	x_3	y_3			

В этой таблице

$$\Delta y_0 = y_1 - y_0; \Delta y_1 = y_2 - y_1; \Delta y_2 = y_3 - y_2; \quad \Delta^2 y_0 = \Delta y_1 - \Delta y_0;$$
$$\Delta^2 y_1 = \Delta y_2 - \Delta y_1; \quad \Delta^3 y_0 = \Delta^2 y_1 - \Delta^2 y_0.$$

Предположим, что шаг в этой таблице – постоянный, т. е. $x_{i+1} - x_i = h_i = h = const \quad i = \overline{0,3}$. Первый ИПН будет выглядеть таким образом:

$$Q_3(x) = y_0 + q_1 \Delta y_0 + \frac{q_1(q_1-1)}{2!}\Delta^2 y_0 + \frac{q_1(q_1-1)(q_1-2)}{3!}\Delta^3 y_0. \text{ Здесь } q_1 = \frac{x - x_0}{h}.$$

Общий вид ИПН степени *n*-й:

$$Q_n^I(x) = y_0 + q_1 \Delta y_0 + \frac{q_1(q_1-1)}{2!}\Delta^2 y_0 + \frac{q_1(q_1-1)(q_1-2)}{3!}\Delta^3 y_0 + \frac{q_1(q_1-1)(q_1-2)(q_1-3)}{4!}\Delta^4 y_0 +$$
$$+ \ldots + \frac{q_1(q_1-1)(q_1-2)\ldots(q_1-n+1)}{n!}\Delta^n y_0.$$

«*I*» означает, что это первый интерполяционный полином Ньютона. Он удобен для вычислений, когда точка x^*, в которой нужно вычислить значение функции, расположена ближе к началу таблицы.

Если точка x^* расположена ближе к концу таблицы, то удобнее пользоваться формулой второго интерполяционного полинома Ньютона:

$$Q_n^{II}(x) = y_n + q_2 \Delta y_{n-1} + \frac{q_2(q_2+1)}{2!}\Delta^2 y_{n-2} + \frac{q_2(q_2+1)(q_2+2)}{3!}\Delta^3 y_{n-3} + \ldots$$
$$+ \frac{q_2(q_2+1)(q_2+2)\ldots(q_2+n-1)}{n!}\Delta^n y_0 \quad \text{, где } q_2 = \frac{x - x_n}{h}.$$

Результаты применения обеих формул одни и те же, если используются одни и те же узлы таблицы. Приведем пример применения формулы первого ИПН.

Пример 4.3. Функция задана в виде таблицы

x	2	4	6	8
y	4	18	32	64

Вычислить значение функции в точке $x^* = 2.8$.

Решение. Так как шаг в таблице постоянный ($h = 2$), то на первом этапе решения задачи интерполяции можно строить и ИПЛ, и ИПН. Воспользуемся формулой интерполяционного полинома Ньютона. Сначала составим таблицу конечных разностей:

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$
2	4	14	0	18
4	18	14	18	
6	32	32		
8	64			

Воспользуемся формулой первого ИПН:

$$Q_3(x) = y_0 + q_1\Delta y_0 + \frac{q_1(q_1-1)}{2!}\Delta^2 y_0 + \frac{q_1(q_1-1)(q_1-2)}{3!}\Delta^3 y_0.\ q_1 = \frac{x-x_0}{h} = \frac{2.8-2}{2} = 0.4.$$

$$Q_3(2.8) = 4 + 0.4\cdot 14 + \frac{0.4(0.4-1)}{2}\cdot 0 + \frac{0.4(0.4-1)(0.4-2)}{6}\cdot 18 = 9.6 + 3*0.24*1.6 = 10.752.$$

При увеличении числа точек таблицы степень полинома тоже возрастает, что приводит к тому, что у интерполяционной кривой могут возникать осцилляции, например такие, как на рис. 4.3.

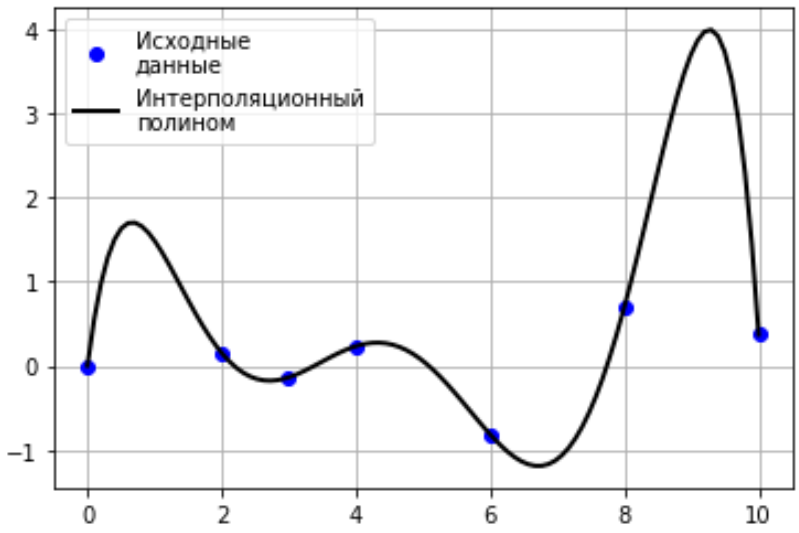

Рис. 4.3. Интерполяция полиномом 6-й степени

Возрастает и количество вычислений. Кроме того, за пределами таблицы полиномы имеют тенденцию сильно убывать или возрастать. Все это свидетельствует о том, что если число узлов интерполяции велико, то построение интерполяционного полинома, проходящего через все точки таблицы, не всегда целесообразно. Для уменьшения погрешности часто поступают следующим образом: на каждом из участков $[x_i, x_{i+1}]$ функцию заменяют сплайном. *Сплайном* называется кусочно-полиномиальная функция, определенная на отрезке $[x_0, x_n]$ и имеющая на этом отрезке некоторое количество непрерывных производных. Чаще всего ограничиваются кубическими сплайнами.

Запишем уравнение кубического сплайна в виде

$$S_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3,\ i = \overline{1,\ n},$$

где x_i – узлы интерполяции; $S_1(x)$ – сплайн, проходящий через точки (x_0, y_0) и (x_1, y_1); $S_n(x)$ – сплайн, проходящий через точки (x_{n-1}, y_{n-1}), (x_n, y_n). Очевидно, что $S_i(x_{i-1}) = a_i$.

Для того чтобы найти коэффициенты всех сплайнов (а их будет n: для трех точек – два, для четырех – три и т. д.), необходимо получить $4n$ уравнений, поскольку каждый сплайн содержит 4 неизвестных коэффициента. Так как каждый сплайн проходит через два узла таблицы, имеем $2n$ уравнений:

$$S_1(x_0) = y_0;\ S_1(x_1) = y_1;\ S_2(x_1) = y_1;\ S_2(x_2) = y_2;\ \cdots;\ S_n(x_{n-1}) = y_{n-1};\ S_n(x_n) = y_n.$$

Еще $2n$-2 уравнения можно получить, приравнивая значения первых и вторых производных соседних сплайнов во внутренних узлах:

$$S_1'(x_1) = S_2'(x_1);\ S_2'(x_2) = S_3'(x_2);\ S_3'(x_3) = S_4'(x_3);\ \cdots;\ S_{n-1}'(x_{n-1}) = S_n'(x_{n-1}),$$
$$S_1''(x_1) = S_2''(x_1);\ S_2''(x_2) = S_3''(x_2);\ S_3''(x_3) = S_4''(x_3);\ \cdots;\ S_{n-1}''(x_{n-1}) = S_n''(x_{n-1}).$$

Два последних уравнения для поиска коэффициентов a_i, b_i, c_i, d_i можно получить, задавая значения первой и/или второй производных в точке x_0 или x_n. Например, можно положить $S_1''(x_0) = 0,\ S_n''(x_n) = 0$. Покажем, как можно построить кубические сплайны для таблицы из трех точек.

Пример 4.4. Для функции, заданной в виде таблицы, построить два кубических сплайна: первый – на отрезке [1, 2], а второй – на отрезке [2, 4].

x	1	2	4
y	8	12	6

Решение. Обозначим первый сплайн $S_1(x)=a+b\cdot x+c\cdot x^2+d\cdot x^3$, второй – $S_2(x)=a_1+b_1\cdot x+c_1\cdot x^2+d_1\cdot x^3$. Первый сплайн проходит через две точки (x_0, y_0) и (x_1, y_1), второй – через точки (x_1, y_1) и (x_2, y_2).

Имеем 4 уравнения: $S_1(x_0)=y_0$; $S_1(x_1)=y_1$; $S_2(x_1)=y_1$; $S_2(x_2)=y_2$.

Вычислим первые и вторые производные от каждого сплайна:

$$S_1'(x) = b + 2cx + 3dx^2,\ S_2'(x) = b_1 + 2c_1x + 3d_1x^2,\ S_1''(x) = 2c + 6dx,\ S_2''(x) = 2c_1 + 6d_1x.$$

Приравняв их значения в точке x_1, получим еще два уравнения:

$$S_1'(x_1) = S_2'(x_1),\ S_1''(x_1) = S_2''(x_1).$$

Последние два уравнения можно, например, получить, приравняв к нулю первые производные в первой и последней точках таблицы: $S_1'(x_0) = 0,\ S_2'(x_2) = 0.$

В итоге получаем систему из 8 уравнений с 8 неизвестными. В соответствии с введенными обозначениями получилась система

$$a + bx_0 + cx_0^2 + dx_0^3 = y_0$$
$$a + bx_1 + cx_1^2 + dx_1^3 = y_1$$
$$a_1 + b_1x_1 + c_1x_1^2 + d_1x_1^3 = y_1$$
$$a_1 + b_1x_2 + c_1x_2^2 + d_1x_2^3 = y_2$$
$$b + 2cx_1 + 3dx_1^2 - b_1 - 2c_1x_1 - 3d_1x_1^2 = 0$$
$$2c + 6dx_1 - 2c_1 - 6d_1x_1 = 0$$
$$b + 2cx_0 + 3dx_0^2 = 0$$
$$b_1 + 2c_1x_2 + 3d_1x_2^2 = 0$$

Подставляя в систему уравнений исходные данные, получаем

$$a+b+c+d=8$$
$$a+2b+4c+8d=12$$
$$a_1+2b_1+4c_1+8d_1=12$$
$$a_1+4b_1+16c_1+64d_1=6$$
$$b+4c+12d-b_1-4c_1-12d_1=0$$
$$2c+12d-2c_1-12d_1=0$$
$$b+2c+3d=0$$
$$b_1+8c_1+48d_1=0$$

Решение этой системы линейных уравнений дает значения искомых коэффициентов сплайнов. Решить эту систему уравнений можно, например, с помощью функции solve() модуля linalg:

```
import numpy as np
from scipy.linalg import *
A = np.array([
   [1,1,1,1,0,0,0,0],
   [1,2,4,8,0,0,0,0],
   [0,0,0,0,1,2,4,8],
   [0,0,0,0,1,4,16,64],
   [0,1,4,12,0,-1,-4,-12],
   [0,0,2,12,0,0,-2,-12],
   [0,1,2,3,0,0,0,0],
   [0,0,0,0,0,1,8,48]])
b = np.array([8,12,12,6,0,0,0,0]).reshape(8,1)
x=solve(A,b)
for i in range(len(b)):
  print("x(%i)=%.7f" % (i+1,x[i]))
```

Результат:

```
x(1)=23.0000000
x(2)=-35.5000000
x(3)=26.0000000
x(4)=-5.5000000
x(5)=-38.0000000
x(6)=56.0000000
x(7)=-19.7500000
x(8)=2.1250000
```

Для уменьшения длины кода в программе вместо переменных a, b, c, d, a_1, b_1, c_1, d_1 использовались переменные x_i, $i = 1,2, \dots, 8$.

Подставляя полученные значения в уравнения сплайнов, получаем $S_1(x)=23-35.5x+26x^2-5.5x^3$, $S_2(x)=-38+56x-19.75x^2+2.125x^3$.

Построенные сплайны удовлетворяют следующим условиям: каждый проходит через две соседние точки таблицы, в точке «стыка» (т. е. в точке x_1) значения первых и вторых производных от каждого сплайна совпадают, первая производная от первого сплайна в точке x_0 и первая производная от второго сплайна в точке x_2 равны нулю. Графики этих сплайнов представлены на рис. 4.4.

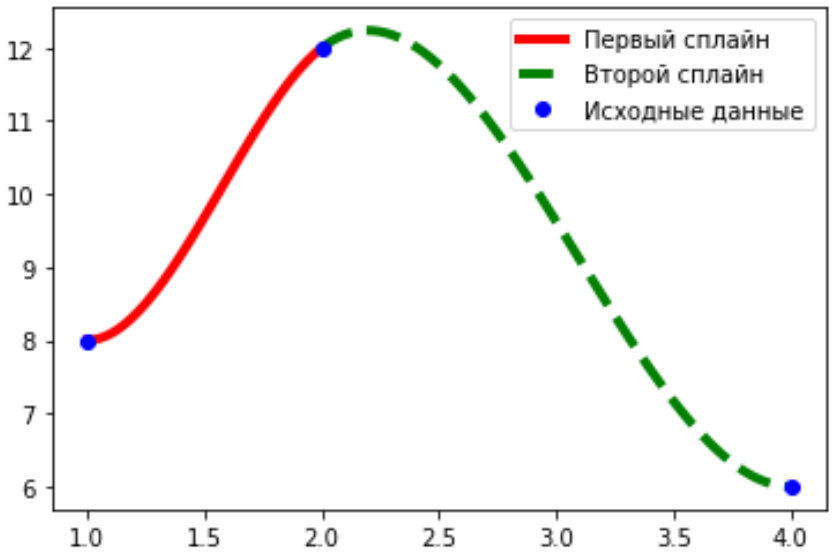

Рис. 4.4. Графики двух кубических сплайнов

4.2. Решения задачи интерполяции как задачи аппроксимации

Поскольку задача интерполяции является частным случаем задачи аппроксимации, уравнение кривой в Python можно получить так же, как это было сделано в гл. 3: с помощью функций curve_fit() и leastsq() модуля optimize библиотеки SciPy. Для этого необходимо, чтобы число неизвестных коэффициентов искомой кривой $Y=f(x,c_0,c_1,\ldots,c_k)$ совпадало с количеством узлов таблицы. В этом случае кривая пройдет через все точки таблицы, а не рядом с ними, как было при решении задачи аппроксимации. Применение этих функций для решения задачи интерполяции проиллюстрируем на примере.

Пример 4.5. Функция задана в виде таблицы

x	1	2	3
y	2	3	−2

Вычислить значение функции в точке $x^* = 1.5$. Решить задачу интерполяции в предположении, что x и y связаны зависимостью $Y(x) = ax\cos(bx) + \frac{c}{x}$. Задачу решить с помощью функций leastsq()

и curve_fit() модуля optimize. Дать графическое решение задачи интерполяции.

Решение. Код программы с использованием функции leastsq():

```
import numpy as np
from scipy import *
import matplotlib.pyplot as plt
from scipy import optimize
x=np.array([1,2,3]);y=np.array([2,3,-2])
def f(t,a,b,c):
    return a*t*np.cos(b*t)+c/t
# определяем невязки
def g(x0):
    return y- f(x, *x0)
x0_init = (1, 1, 1) # задаем начальное приближение
# передаем в функцию leastsq требуемые параметры
x_opt, _ = optimize.leastsq(g, x0_init)
a, b, c =x_opt[0], x_opt[1], x_opt[2]
v=np.sum((f(x, a, b, c)-y)**2)
print("Коэффициенты кривой:")
print("a= %.6f" % a);print("b= %.6f" % b);print("c= %.6f" % c)
print("Сумма площадей квадратов отклонений равна  %.6f" % v)
print("Уравнение кривой Y=(%.6f)x*cos(%.6fx)+(%.6f)/x"
 % (a, b, c))
p=1.5;q=f(p,a,b,c);
print("Значение функции в точке %.1f равно %.6f" % (p,q))
plt.plot(x, y, 'bo', label="Исходные\nданные")
x1=np.linspace(min(x),max(x),100)
plt.plot(x1, f(x1, a, b, c),
        label="Интерполяционная\nкривая",c='r')
plt.plot(p, q,"*g",ms=10,label="Решение")
plt.grid()
plt.legend()
plt.show()
```

Решение:

```
Коэффициенты кривой:
a= 4.435928
b= 0.559062
c= -1.760571
```

Сумма площадей квадратов отклонений равна 0.000000.
Уравнение кривой: Y=(4.435928)x*cos(0.559062x)+(-1.760571)/x.
Значение функции в точке 1.5 равно 3.274478.

Графическое решение задачи представлено на рис. 4.5.

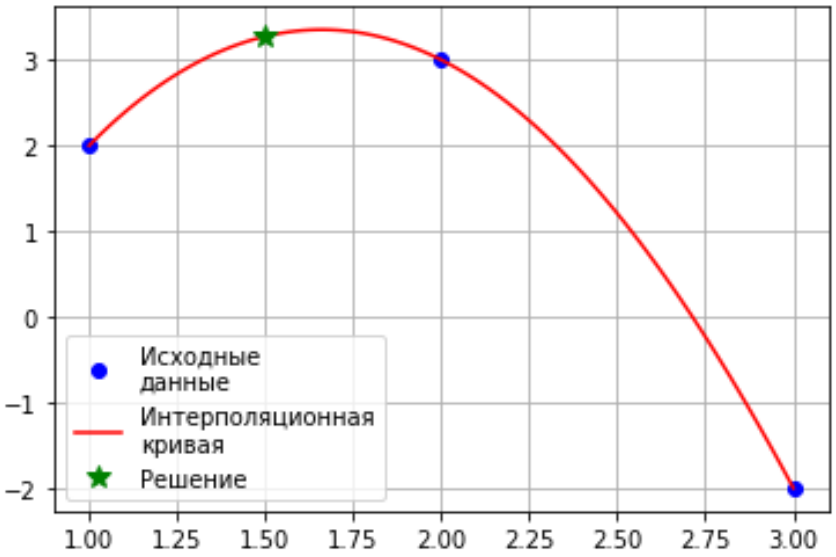

Рис. 4.5. Графическое решение примера 4.5

При решении задачи с помощью функции curve_fit() изменения в коде будут минимальными:

```
args,_  = optimize.curve_fit(f, x, y,[1,1,1])
a, b, c= args[0], args[1], args[2]
v=np.sum((f(x,a,b,c)-y)**2)
```

Решение задачи с использованием функции curve_fit() идентично решению задачи с использованием функции leastsq().

4.3. Работа с модулем scipy.interpolate

SciPy предоставляет специальные средства для решения одномерных и многомерных задач интерполяции.

Для интерполяции на плоскости предусмотрена функция interp1d(). Синтаксис функции:

interp1d (x, y, kind='linear' , axis =-1 , copy = True , bounds_error = None, fill_value = nan , accept_sorted = False).

Здесь *x* и *y* – исходные данные, массивы значений, используемых для аппроксимации некоторой функции *f*. Количество элементов в массивах должно быть одинаковым.

Другие необязательные параметры функции interp1d():

kind – строка или целое число, указывающие тип интерполяции. Если параметр задается в виде строки, то он может принимать одно из значений: 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous' или 'next'. Значения 'zero', 'slinear', 'quadratic' и 'cubic' относятся к сплайн-интерполяции нулевого, первого, второго или третьего порядка. Если значения параметра равны 'previous' или 'next', то возвращается предыдущее или следующее значение функции *y*; 'nearest-up' и 'nearest' различаются при интерполяции нецелых чисел (например, 0,5, 1,5) тем, что 'nearest-up' округляет значения с избытком, а 'nearest' – с недостатком.

На рис. 4.6 приведены графики интерполяционных полиномов при разных значениях параметра *kind* для одних и тех же значений *x* и *y*.

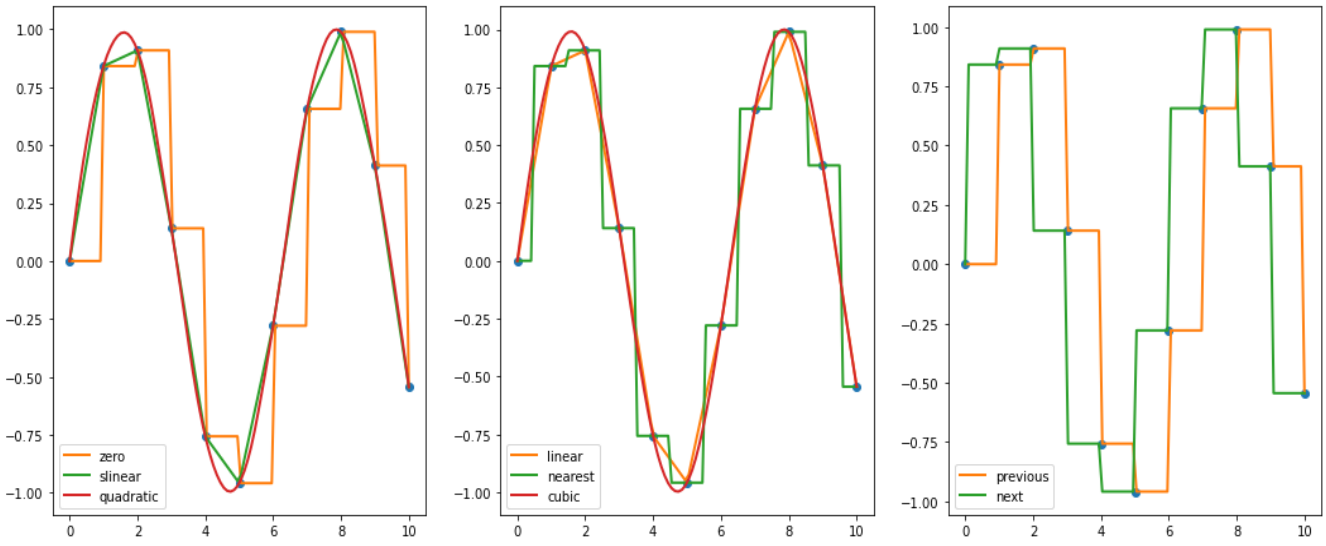

Рис. 4.6. Решение задачи интерполяции при разных значениях параметра kind

При значениях параметра *kind*, равных 'zero' и 'nearest', имеем ступенчатую интерполяцию, при значениях 'linear' и 'slinear' – линейную интерполяцию, при которой все соседние точки таблицы соединяются прямой, при значениях 'quadratic' и 'cubic' строятся сплайны второго и третьего порядка. Кривые высших порядков можно создавать, напрямую задавая значения степени полинома, например таким образом: kind=7.

Остальные параметры функции interp1d():

axis – целочисленная переменная. Определяет ось, вдоль которой выполняется интерполяция. По умолчанию используется последняя ось;

copy – логическая переменная. Если значение параметра равно True, создаются внутренние копии *x* и *y*. Если значение параметра равно False, используются ссылки на *x* и *y*;

bounds_error – булева переменная. При значении переменной, равном True, возникает ошибка ValueError всякий раз, когда предпринимается попытка вычисления значения функции в точке, выходящей за пределы диапазона изменения переменной *x*. Если значение переменной *bounds_error* равно False (или 0), результатом работы функции является NaN, если только значение параметра *fill_value* не равно "extrapolate". Если значение параметра не задано, то возникает ошибка, если только значение *fill_value* не равно "extrapolate";

fill_value – параметр, использующийся при решении задачи экстраполяции. Если *fill_value*="extrapolate", то решение будет получено и в случае, когда значение аргумента, при котором вычисляется значение функции, находится за пределами интервала [min(*x*), max(*x*)];

accept_sorted – логическая переменная. Если *accept_sorted*=False, значения *x* могут быть расположены в любом порядке, и затем они автоматически сортируются. Если значение параметра равно True, *x* должен быть массивом монотонно возрастающих значений.

Рассмотрим решение задачи интерполяции с помощью функции interp1d() на простом примере.

Пример 4.6. Функция задана в виде таблицы

x	1	2	4
y	0	6	24

Вычислить значение функции в точке $x^* = 3$. Решить задачу интерполяции с помощью функции interp1d(), используя линейную и квадратичную интерполяцию. Дать графическое решение задачи.

Решение:

```
from scipy import interpolate; import matplotlib.pyplot as plt
import numpy as np
x= np.array([1,2,4]);y=np.array([0,1,24])
f=interpolate.interp1d( x , y)
f1=interpolate.interp1d( x , y,'quadratic')
xnew=3;ynew = f(xnew);ynew1 = f1(xnew)
```

```
fig, ax=plt.subplots()
x1=np.linspace(1,4,10)
plt.plot(x1,f(x1),'r',x1,f1(x1),'g--',lw=4);plt.plot(x,y,'ob')
label=["Линейная интерполяция", "Квадратичная интерполяция",
 "Исходные данные","Решение"]
plt.plot(xnew,ynew,'*k',xnew,ynew1,'*k',ms=10)
plt.legend(label);plt.show()
print("Значение функции в точке %.1f  при линейной интерполяции равно %.3f"  % (xnew,ynew))
print("Значение функции при квадратичной интерполяции равно %.3f" % ynew1)
```

Результат:

Значение функции в точке 3.0 при линейной интерполяции равно 12.500.
Значение функции при квадратичной интерполяции равно 9.000.

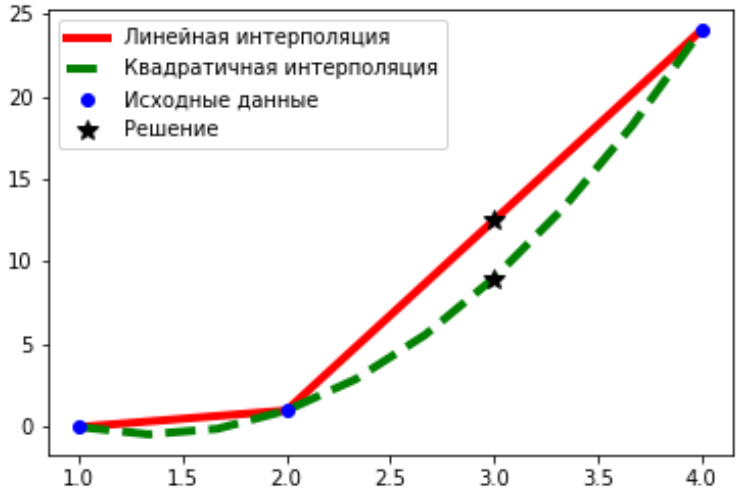

Рис. 4.7. Решение примера 4.6 при линейной и квадратичной аппроксимации

При решении задачи с помощью функции interp1d() сначала определяется отрезок, на котором находится точка, в которой необходимо вычислить значение функции ($x^* = 3$). В условиях примера 4.6 это отрезок [2, 4]. При линейной интерполяции на этом отрезке строится прямая, проходящая через точки с координатами (2, 1) и (4, 24). Уравнение прямой $y(x)=11.5x-22$. В это уравнение подставляется точка $x^* = 3$. $y(3)=12.5$. При квадратичной аппроксимации строились два сплайна 2-й степени. Их уравнения в данном примере при трех узлах интерполяции на обоих участках совпадают: $y(x)=6-9.5x+3.5x^2$, $y(3)=9$.

В последнее время в вычислительной практике широкое распространение получили *B-сплайны* (*Basic spline*). Они используются как для интерполяции функций, так и в качестве базисных функций при построении методов типа конечных элементов. Метод конечных элементов – это численный метод решения дифференциальных уравнений счастными производными, а также интегральных уравнений, возникающих, в частности, при решении задач прикладной физики. В данном пособии мы не расматриваем теоретические основы построения *B*-сплайнов, а ограничимся лишь их применением к решению задач аппроксимации.

Для интерполяции с помощью *B*-сплайнов может быть использована функция splrep(). В отличие от обычной сплайн-интерполяции, «сшивка» элементарных *B*-сплайнов производится не в точках (x_i, y_i), а в других точках, координаты которых в функции splrep() могут либо вычисляться автоматически, либо задаваться пользователем. Таким образом, требование равномерного следования узлов при интерполяции *B*-сплайнами отсутствует, и ими можно приближать разрозненные данные.

Синтаксис функции splrep(): splrep(x, y, w=None, xb=None, xe=None, k=3, task=0, s=None, t=None, full_output=0, per=0, quiet=1). Необходимыми параметрами этой функции являются массивы исходных данных *x* и *y*. Элементы вектора *x* должны быть расположены в возрастающем порядке. Результатом работы функции является кортеж (*t*, *c*, *k*), содержащий узловые точки *t*, которые используются для построения *B*-сплайна, его коэффициенты и порядок сплайна *k*. Порядок сплайнов по умолчанию – кубический, но его можно изменить с помощью входного параметра *k*. Число точек таблицы (len(x)) должно быть больше *k*.

Параметры функции splrep():

w – строго положительный массив весов той же длины, что и *x* с *y*. Веса используются при вычислении коэффициентов «взвешенного» сплайна методом наименьших квадратов. Если погрешности в значениях *y* имеют стандартное отклонение, заданное вектором *d*, то вес *w* должен быть равным 1/*d*. Значение по умолчанию ones(len(x)): все веса равны 1;

xb, xe – переменная типа float. Интервал интерполяции. Если параметр опущен, по умолчанию используются значения x[0] и x[-1] соответственно;

k – переменная типа int. Степень сплайна;

task – параметр, принимающий одно из значений {1, 0, -1}. Если *task*=0, находятся *t* и *c* для заданного коэффициента сглаживания *s*. Если *task*=1, вычисляются *t* и *c* для предыдущего значения коэффициента сглаживания. До этого должен быть осуществлен вызов функции splrep() с *task*=0 или *task*=1 для того же набора данных. Если *task*=-1, находится «взвешенный» сплайн для заданного набора узлов *t*;

s – переменная типа float. Коэффициент сглаживания. Величина гладкости обеспечивает выполнение условий sum((w*(y-g))**2,axis=0)≤ *s*, где g(x) – результат интерполяция набора данных (*x*,*y*). По умолчанию *s*=m-sqrt(2*m), если указаны веса. Здесь *m* – число точек таблицы. Значение параметра *s*=0, если вес не указан;

t – массив, с помощью которого задаются узлы в случае, когда значение параметра *task*=-1;

full_output – булева переменная. Если значение отлично от 0, выводятся дополнительные выходные параметры функции splrep();

per – булева переменная. Если значение не 0, исходная функция считается периодической с периодом x[m-1] - x[0];

quiet – булева переменная. Если значение не равно 0, подавляется вывод необязательных сообщений.

После построения сплайна вычислить его значения в заданных точках можно с помощью функции splev(). Ее синтаксис:

splev(x, tck, der=0, ext=0).

Аргументами функции являются:

x – массив точек, в которых нужно вычислить значение сглаженного сплайна или его производных;

tck – кортеж или объект *B*-Spline. Если *tck* – кортеж, то это последовательность длиной 3, возвращаемая функциями splrep() или splprep(). *tck* содержит узлы, коэффициенты и степень сплайна;

der – значение типа int. Порядок производной сплайна для вычисления ее значений в заданных точках *x* (должен быть меньше или равен степени сплайна *k*);

ext – значение типа int, задающее порядок действий в случае, когда значение *x* выходит за пределы [x[0], x[-1]]. В зависимости от значения параметра *ext* функция возвращает экстраполированное значение (*ext*=0), 0 (*ext*=1), сообщение ValueError() (*ext*=2), граничное значение (y[0] или y[-1]) (*ext*=3). Значение параметра *ext* по умолчанию – 0.

Результатом работы функции splev() являются значения вычисляемой функции *y* – массив ndarray или список массивов ndarrays.

Функция spalde() вычисляет производные полученного сплайна в любой точке, функция splint() – интеграл от сплайна с пределами интегрирования, принадлежащими отрезку [*xb*, *xe*]. Кроме того, для кубических сплайнов (*k*=3) можно вычислить корни сплайна (функция sproot()). Работу этих функций рассмотрим на примерах.

Пример 4.7. По данным таблицы построить графики *B*-сплайнов 2-й и 3-й степени на отрезке [1, 5]:

x	1	2	4	5
y	8	12	6	3

Решение:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
x = np.array([1,2,4,5]); y = np.array([8,12,6,3])
t2 = interpolate.splrep(x, y,k=3)
t1 = interpolate.splrep(x, y,k=2)
xnew = np.linspace(1, 5, 100)
ynew = interpolate.splev(xnew, t2, der=0)
ynew1 = interpolate.splev(xnew, t1, der=0)
plt.figure()
plt.plot(x, y, 'x', xnew, ynew1,'--' ,xnew, ynew,lw=2)
plt.legend(['Исходные данные','Сплайн 2 степени', 'Сплайн 3 степени'])
plt.show()
```

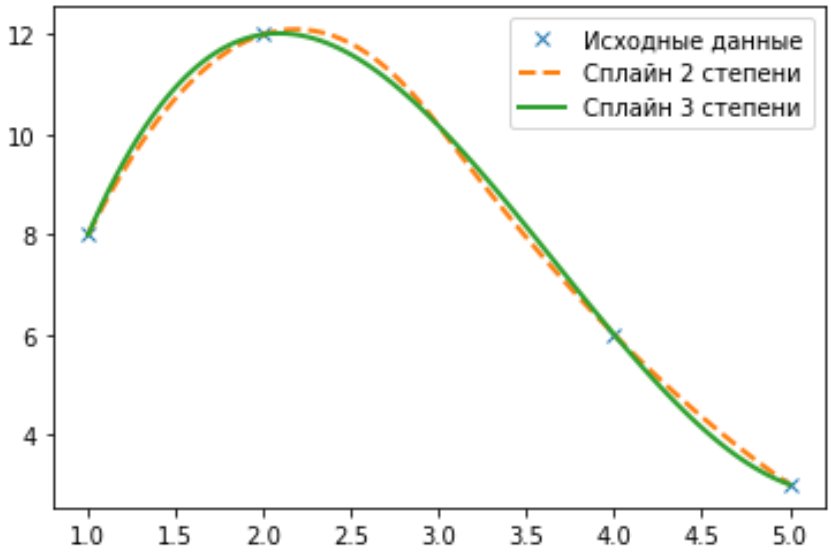

Рис. 4.8. Интерполяция B-сплайнами

Краткий комментарий. Создаются два объекта: сплайны третьей (t2) и второй (t1) степени. С помощью функции splev() вычисляются их значения в 100 точках отрезка [1, 5] и строятся графики сплайнов.

Если вывести на экран, например, значение t2, получим

(array([1., 1., 1., 1., 5., 5., 5., 5.]), array([8. , 18.77777778, 4.66666667, 3. , 0. , 0. , 0. , 0.]), 3).

Здесь 1 и 5 – те значения аргументов, которые функция splrep() выбрала для построения сплайнов. В данном случае она построила один сплайн. Второй массив – это массив коэффициентов *B*-сплайна. 3 – это степень сплайна. Полученные коэффициенты относятся к классической форме записи уравнения *B*-сплайна. Оно достаточно громоздко. Покажем, как могут быть найдены уравнения сплайнов в более привычном виде: $y = c_0 + c_1 x + c_2 x^2 + c_3 x^3$. Чтобы получить значения c_0, c_1, c_2 и c_3 в случае интерполяции кубическими *B*-сплайнами, составим и решим на каждом участке интерполяции систему 4 линейных уравнений с 4 неизвестными, как это было сделано в примере 4.1. Для получения этих уравнений выберем любые 4 (по числу неизвестных коэффициентов) значения *x* на каждом участке интерполяции и вычислим значения сплайна в этих точках с помощью функции splrep(). Покажем, как можно найти уравнения сплайнов на следующем примере.

Пример 4.8. Найти уравнения кубических *B*-сплайнов, полученных с помощью функции splrep() для таблицы

x	1	2	4	5	7	10	11
y	0	1	24	6	8	12	1

Решение:

```
import matplotlib.pyplot as plt; from scipy import interpolate
import numpy as np; import scipy.linalg
x= np.array([1,2,4,5,7,10,11]);y=np.array([0,1,24,6,8,12,1])
t1 = interpolate.splrep(x, y); rr=np.unique(t1[0])
def f(xx):
  yd=interpolate.splev(xx, t1, der=0)
  yy=yd.reshape(4,1)
  A=xx[:,np.newaxis]**[0,1,2,3]
  c=scipy.linalg.solve(A,yy)
  x1=np.linspace(min(xx),max(xx),100)
  ym=c[0]+c[1]*x1+c[2]*x1**2+c[3]*x1**3
```

```
  plt.plot(x1,ym,'--',lw=3)
  print("Интервал интерполяции [%i, %i]" % (a1[0],a1[1]))
  for i in range(len(xx)):
   print("Коэффициент c(%i)=%.6f" % (i,c[i]))
  print("Уравнение B-сплайна y=%.6f+(%.6f)x+\
    (%.6f)x^2+(%.6f)x^3" % (c[0],c[1],c[2],c[3]))
  return c
a1=np.zeros(2)
for i in range(len(rr)-1):
  a1[0]=rr[i];  a1[1]=rr[i+1]
  xx=np.random.uniform(rr[i],rr[i+1],4)
  f(xx)
x1=np.linspace(min(x),max(x),100)
y3 = interpolate.splev(x1, t1, der=0)
plt.plot(x1,y3,'k',lw=1);plt.plot(x,y,'*k',ms=10)
plt.grid(); plt.show()
```

Из результатов программы приведем лишь уравнения *B*-сплайнов и графики полученных кривых.

Интервал интерполяции [1, 4].

Уравнение *B*-сплайна: y=43.079187+(-74.388578)x+ (35.944289)x^2+(-4.634898)x^3.

Интервал интерполяции [4, 5].

Уравнение *B*-сплайна: y=-956.522803+(675.312915)x+ (-151.481084)x^2 +(10.983883)x^3.

Интервал интерполяции [5, 7].

Уравнение *B*-сплайна: y=724.199562+(-333.120504)x+ (50.205600)x^2+(-2.461896)x^3.

Интервал интерполяции [7, 11].

Уравнение *B*-сплайна: y=-42.364376+(-4.593102)x+ (3.273114)x^2+(-0.227016)x^3.

Комментарий к программе. Вводим исходные данные и создаем объект t1 с информацией о *B*-сплайне. Из массива коэффициентов выбираем уникальные значения (rr). Их 5: [1., 4., 5., 7., 11.]. Это концы интервалов, на которых построены сплайны. Создаем функцию *f*(*xx*), аргументом которой является массив с узлами интерполяции, а результатом работы – вектор искомых коэффициентов *c*. Вычисляем значение сплайн-функции в узлах интерполяции (*yd*). Вектор-столбец из этих значений – правая часть системы уравнений *Ac*=*y*. Создаем матрицу *A*

с левой частью системы уравнений и решаем получившуюся систему с помощью функции scipy.linalg.solve(). Получаем вектор *c* искомых коэффициентов. Для построения графиков полинома с полученными коэффициентами формируем массивы с координатами точек графика *x* и *y*: (x1, ym). Строим график на соответствующем интервале пунктирной линией толщиной 3. Выводим на экран интервал, на котором строился полином, коэффициенты и уравнение полинома. Далее в цикле по количеству интервалов (их в нашей задаче 4) формируем концы каждого интервала (массив a1), генерируем нужное число точек из каждого интервала (их тоже 4 – по количеству коэффициентов сплайна) и передаем в функцию *f*(*xx*) для построения очередного полинома. Получаем 4 (в нашем примере) полинома. Для проверки правильности решения строим в том же графическом окне график сплайна, полученного с помощью функций splrep() и splev(). График строим на интервале [1, 11]. На график наносим исходные данные и сетку. Выводим график в отдельное графическое окно. Видим, что графики совпали, т. е. мы правильно нашли коэффициенты сплайнов.

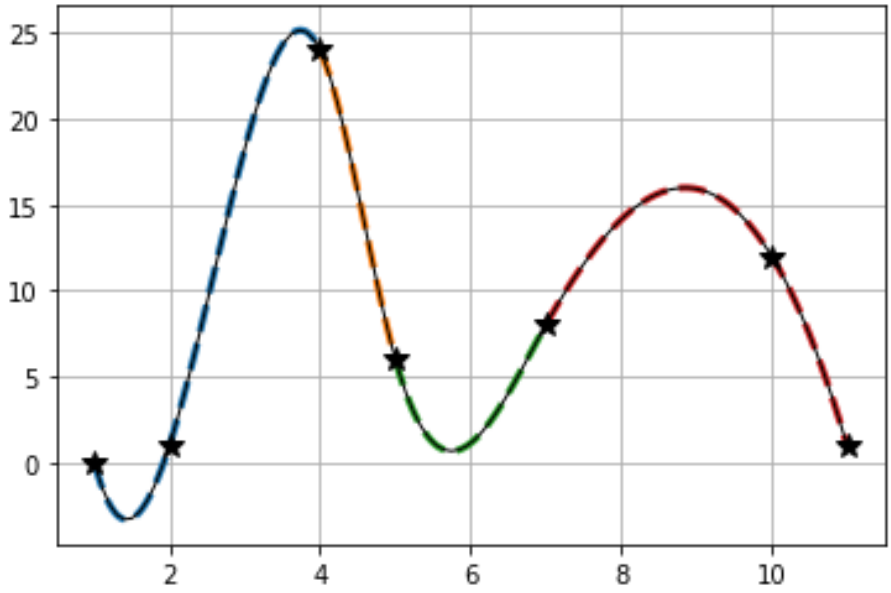

Рис. 4.9. Графики четырех B-сплайнов

Как правило, на практике нас обычно интересуют лишь значения таблично заданной функции в некоторых точках, а не уравнения сплайнов, поэтому данный пример приведен исключительно в иллюстративных целях.

Если нас интересуют корни сплайнов, построенных с помощью функции splrep(), то их можно найти с помощью функции sproot(). Покажем, как это можно сделать.

Пример 4.9. Найти корни *B*-сплайна 3-й степени, построенного по таблице

x	4	5	7	9
y	−6	3	−2	6

Решение:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
x = np.array([4,5,7,9])
y = np.array([-6,3,-2,6])
t1 = interpolate.splrep(x, y,k=3)
xnew = np.linspace(4, 9, 50)
ynew = interpolate.splev(xnew, t1, der=0)
plt.figure()
plt.plot(x, y, 'x', xnew, ynew)
plt.legend(['Исходные данные', 'Сплайн'])
plt.hlines(0,x[0],x[-1],color='g')
plt.title('Кубическая сплайн-интерполяция')
plt.grid()
plt.show()
interpolate.sproot(t1)
```

Результат решения задачи – график (рис. 4.10) и значение корней сплайна.

Значения корней: array([4.48407752, 6.51766416, 8.5097087]).

При необходимости можно вычислить производные от функций, заданных в виде таблицы, т. е. решить задачу численного дифференцирования. Для этого строится сплайн и от него находится соответствующая производная (первая, вторая, третья) в нужных точках. В примере 4.10 показано, как это можно сделать с помощью функции spalde() модуля interpolate.

Пример 4.10. Построить графики первой, второй и третьей производных от кубического *B*-сплайна, построенного по таблице

x	4	5	7	9	12
y	−6	3	−2	6	−1

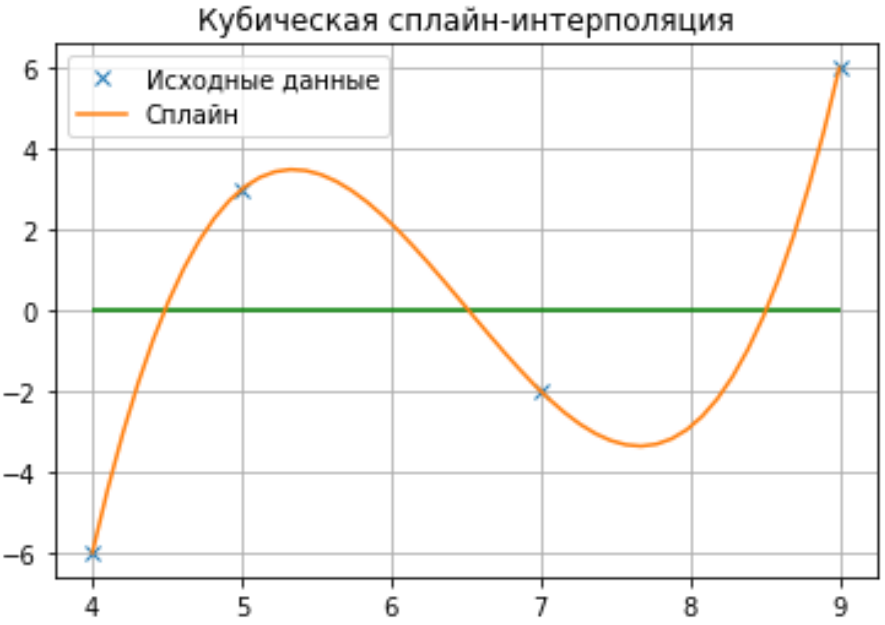

Рис. 4.10. Графическое решение задачи поиска корней сплайна

Решение:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
x = np.array([4,5,7,9,12]); y = np.array([-6,3,-2,6,-1])
t1 = interpolate.splrep(x, y); xnew = np.linspace(4, 12, 50)
yderiv = interpolate.spalde(xnew, t1)
for i in range(1,len(yderiv[0]),1):
  plt.plot(xnew, [d[i] for d in yderiv], '--',
        label=f"{i} производная")
plt.grid()
plt.legend()
plt.title('Все производные B-сплайна')
plt.show()
t1[0]
```

На рис. 4.11 приведены графики производных на интерале [4, 12].

В коде t1[0] *t* – точки, которые используются для построения *B*-сплайна: array([4., 4., 4., 4., 7., 12., 12., 12., 12.]). Фактически мы имеем «склейку» из двух сплайнов. Первый построен на отрезке [4, 7], второй – на отрезке [7, 12]. От этих двух сплайнов и берутся производные.

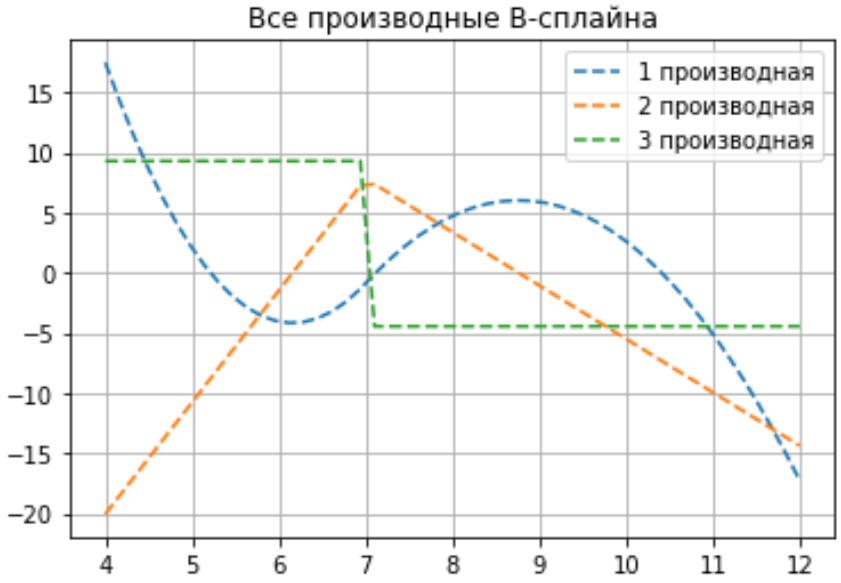

Рис. 4.11. Графики производных от сплайнов примера 4.10

Краткий комментарий. Первая производная от кубического сплайна – это парабола, вторая производная – прямая, а третья – константа.

От полученного сплайна можно найти определенный интеграл, т. е. решить задачу численного интегрирования. В примере 4.11 показано, как это можно сделать.

Пример 4.11. Вычислить интеграл от кубического *B*-сплайна, построенного по таблице

x	1	2	4	5
y	8	12	6	3

Решение:

```
import numpy as np
from scipy import interpolate
x = np.array([1,2,4,5]);y = np.array([8,12,6,3])
t1 = interpolate.splrep(x, y);p=interpolate.splint(x[0], x[-1], t1)
print("Интеграл равен %.5f" % p)
```

Результат:

Интеграл равен 34.44444.

Описанные возможности подбора сплайнов также доступны через объектно-ориентированный интерфейс. Одномерные сплайны являются объектами класса UnivariateSpline и создаются при передаче табличных данных *x* и *y* в качестве аргументов конструктору. Класс позволяет вызывать объект и передавать в него желаемые значения оси *x*, при

которых нужно вычислить значение функции, возвращая интерполированные значения *y*. Методы integral, derivatives и roots также доступны для UnivariateSpline объектов, позволяя вычислять определенные интегралы, производные и корни для сплайна.

Класс UnivariateSpline также можно использовать для сглаживания данных, задавая ненулевое значение параметра сглаживания *s*. В этом случае сплайн уже является не строго интерполирующим сплайном, а скорее сглаживающим. Если это нежелательно, в модуле существует класс InterpolatedUnivariateSpline. Это подкласс UnivariateSpline, который строит сплайны, проходящие через все точки таблицы, что эквивалентно нулевому значению параметра сглаживания.

В качестве примера работы с классом рассмотрим элемент подкласса InterpolatedUnivariateSpline.

Пример 4.12. Вычислить значения функции, заданной таблично, в точках $x^* = 0$ (задача экстраполяции) и $x^* = 6$ (задача интерполяции):

x	1	2	4	7
y	9	12	6	9

Дать графическое решение задачи.

Решение:

```
import numpy as np;import matplotlib.pyplot as plt
from scipy import interpolate
x = np.array([1,2,4,7]);y = np.array([9,12,6,9])
s = interpolate.InterpolatedUnivariateSpline(x, y)
xx=[0,6];yy=s(xx)
for i in range(len(xx)):
  print('Значение функции в точке %i равно %.5f'
        % (xx[i],yy[i]))
xnew = np.arange(0, 7.1, .1);ynew = s(xnew)
plt.plot(x, y, 'x', xnew, ynew,'g',lw=2,ms=13)
plt.plot(xx,yy,'*k',ms=12)
plt.legend(['Данные','Сплайн','Решение задачи'])
plt.show()
```

Результат:

```
Значение функции в точке 0 равно -1.73333.
Значение функции в точке 6 равно 2.66667.
```

Графическое решение задачи представлено на рис. 4.12.

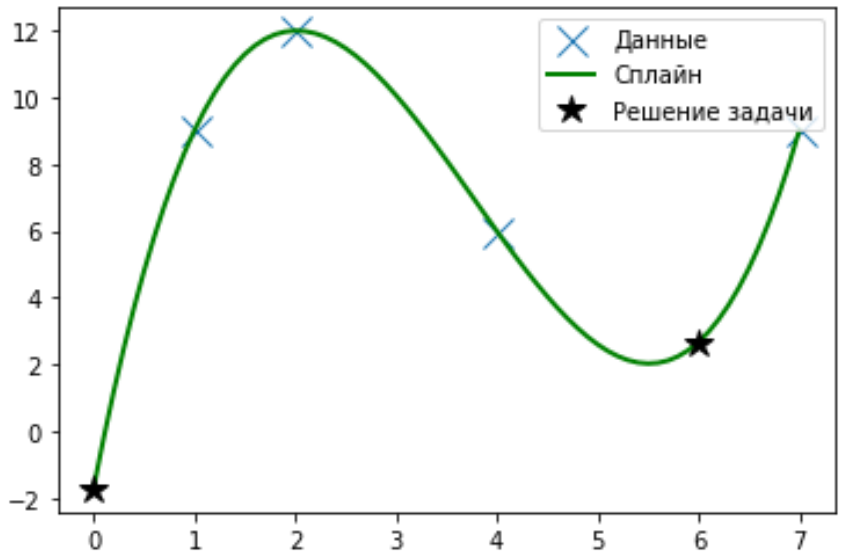

Рис. 4.12. Графическое решение задач экстраполяции и интерполяции

Для решения задачи использовался подкласс InterpolatedUnivariateSpline(x, y). Он имеет следующие параметры:

x – строго возрастающий массив узловых точек;

y – значения функции;

w – массив весовых коэффициентов. Значение по умолчанию None – все веса одинаковые;

bbox – последовательность, указывающая границу интервала аппроксимации. Значение по умолчанию: bbox=[x[0], x[-1]];

k – степень полинома. Целое число от 1 до 5;

der – переменная типа int. Порядок производной сплайна для вычисления ее значений в заданных точках *x* (должен быть меньше или равен степени сплайна *k*);

ext – переменная типа int. Значение, задающее порядок действий в случае, когда значение *x* выходит за пределы [x[0], x[-1]]. Если *ext*=0, возвращает экстраполированное значение, если *ext*=1, возвращает 0, если *ext*=2, возвращает сообщение ValueError, если *ext*=3, возвращает граничное значение (y[0] или y[-1]). Значение параметра *ext* по умолчанию – 0;

check_finite – булева переменная. С помощью нее проверяется, содержат ли входные массивы только конечные числа. Отключение может привести к повышению производительности, но может привести и к проблемам (сбои, незавершение или неправильные результаты), если входные данные содержат значения бесконечности или значения NaN. Значение по умолчанию – False.

В 1985 г. М. Дж. М. Пауэлл предложил метод многомерной интерполяции с радиальной базисной функцией (RBF). Радиальная

базисная функция – это вещественная функция, значение которой зависит только от расстояния от начала координат, т. е. Ф (x) = Ф (‖x‖), или это может быть расстояние до любой точки *c*. Точка *c* называется центральной точкой. Здесь ‖ ‖ – обозначение нормы. Часто используемые радиальные базисные функции: функция Гаусса $\left(e^{-(\varepsilon x)^2}\right)$, мультиквадратичная $\left(\varphi(r)=\sqrt{1+(\varepsilon r)^2}\right)$, обратная квадратичная $\left(\varphi(r)=\frac{1}{1+(\varepsilon r)^2}\right)$, обратная мультиквадратичная $\left(\varphi(r)=\frac{1}{\sqrt{1+(\varepsilon r)^2}}\right)$, полигармонический сплайн $\left(\begin{cases}\varphi(r)=r^k, & k=1,3,5,...\\ \varphi(r)=r^k\ln r, & k=2,4,6,...\end{cases}\right)$, специальный полигармонический сплайн $\left(\varphi(r)=r^2\ln r\right)$. Здесь $r=\|x-x_i\|$. Покажем, как можно построить радиальную базисную функцию, на примере.

Пример 4.13. Функция $f(x)=e^{x\cos(3\pi x)}$ задана на интервале [0, 1]. Разбив интервал интерполяции на 14 равных частей, решить задачу интерполяции с помощью радиальных базисных функций. В качестве радиальных базисных функций выбрать функцию Гаусса $\varphi(r)=e^{-(\varepsilon r)^2}$ с параметром формы ε=3.

Решение. Требуется построить функцию $s(x)=\sum_{i=0}^{14}\omega_i\varphi(\|x-x_i\|)$, где $x_i=\frac{i}{14}, i=\overline{0,14}$. В качестве нормы примем модуль разности $\|x-x_i\|=|x-x_i|, i=\overline{0,14}$. Коэффициенты ω_i найдем из условия $s(x_i)=f(x_i), i=\overline{0,14}$. В матричной форме это можно записать следующим образом:

$$\begin{pmatrix}\varphi(|x_0-x_0| & \varphi(|x_1-x_0| & \cdots & \varphi(|x_{14}-x_0|\\ \varphi(|x_0-x_1| & \varphi(|x_1-x_1| & \cdots & \varphi(|x_{14}-x_1|\\ \vdots & \vdots & \ddots & \vdots\\ \varphi(|x_0-x_{14}| & \varphi(|x_1-x_{14}| & \cdots & \varphi(|x_{14}-x_{14}|\end{pmatrix}\cdot\begin{pmatrix}w_0\\ w_1\\ \vdots\\ w_{14}\end{pmatrix}=\begin{pmatrix}f(x_0)\\ f(x_1)\\ \vdots\\ f(x_{14})\end{pmatrix}.$$

Решая эту систему линейных уравнений, находим неизвестные коэффициенты w_i. Код программы:

```
import matplotlib.pyplot as plt;
from scipy import interpolate;from scipy.linalg import *
```

```
from numpy import *
import numpy as np
n=15; x = np.linspace(0,1,n); y=np.exp(x*np.cos(3*np.pi*x))
A=np.zeros((n,n))
for i in range(n):
  for j in range(n):
    A[i][j]=np.exp(-(abs(x[i]-x[j])*3)**2)
c=solve(A,y)
def s(t):
  z=0
  for i in range(n):
    z+=c[i]*np.exp(-(3*(t-x[i]))**2)
  return z
r=linspace(0,1,150)
yr=np.exp(r*np.cos(3*np.pi*r))
plt.plot(r,s(r),'--r',r,yr,x,y,'ob',lw=3)
plt.legend(['Решение с помощью\nрадиальных базисных функций',
  'Исходная кривая','Данные для построения\nрадиальных \
   базисных функций'])
plt.show()
plt.plot(r,-yr+s(r))
```

Результаты решения представлены на рис. 4.13 и 4.14. На первом из них построены графики исходной кривой ($f(x)=e^{x\cos(3\pi x)}$) и кривой, полученной в результате решения задачи интерполяции. На график нанесены исходные данные.

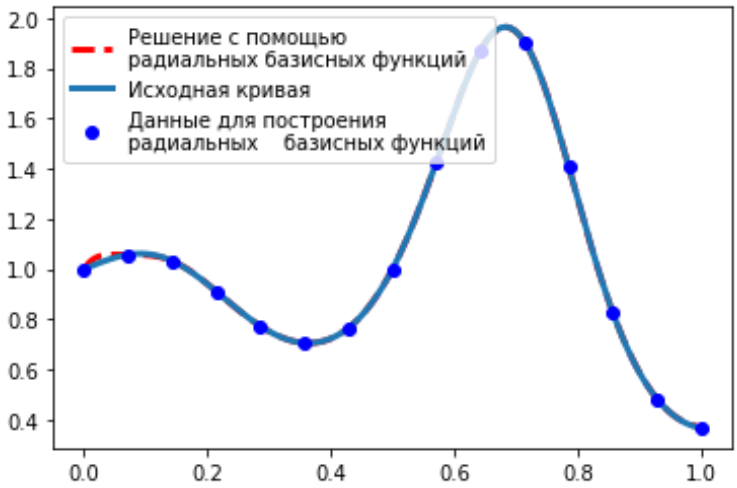

Рис. 4.13. Решение задачи интерполяции с помощью радиальных базисных функций

На втором графике изображена функция $g(x)$, являющаяся разностью между интерполяционной кривой $s(x)$ и исходной функцией $f(x)$: $g(x) = s(x) - f(x)$.

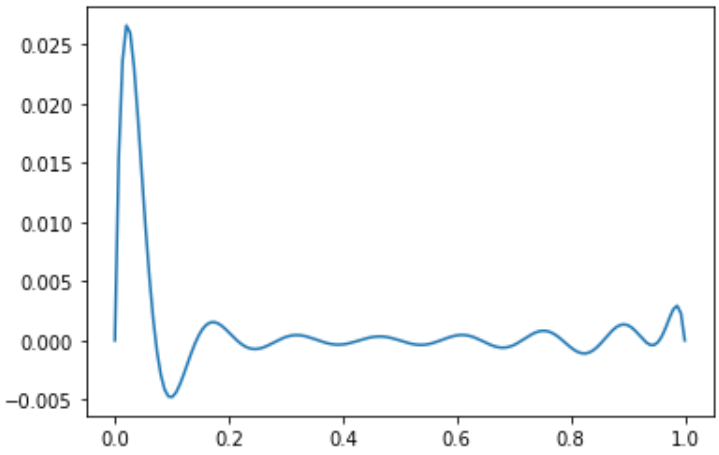

Рис. 4.14. График функции g(x) = s(x) − f(x)

Комментарий к программе. Задаем количество узлов (n = 15) и вычисляем узловые точки и значения исходной функции в этих точках. Формируем матрицу A системы уравнений $Aw=y$. Решаем систему с помощью функции solve() модуля scipy.linalg. Решение системы – вектор c. Формируем функцию $s(t)=\sum_{i=0}^{14} w_i e^{-(3\cdot|t-x_i|^2)}$. Задаем интервал построения графика (переменная r) и строим графики функций в двух графических окнах. Из графиков видно, что интерполяционная и исходная кривые практически совпадают.

Приведем решение примера 4.13 с другой базисной функцией, например с $\left(\varphi(r)=\frac{1}{1+(\varepsilon r)^2}\right)$:

```
import matplotlib.pyplot as plt; from scipy import interpolate
import scipy.linalg
from numpy import *
import numpy as np
def RBF(t):
  h=1/(1+(e*t)**2)
  return h
def s(t):
  z=0
  for i in range(n):
    z+=c[i]*RBF(t-x[i])
```

```
  return z
def g(x):
  y=np.exp(x*np.cos(3*np.pi*x)) # Значение исходной функции
  return y
n=15;e=3; # Количество слагаемых и параметр формы
a1=0;b1=1; # Интервал интерполяции
x = np.linspace(a1,b1,n) # Значение аргумента
A=np.zeros((n,n))
for i in range(n):
  for j in range(n):
    A[i][j]=abs(x[i]-x[j])
B=RBF(A)
c=scipy.linalg.solve(B,g(x));r=linspace(a1,b1,550)
plt.plot(r,s(r),'--',lw=5); plt.plot(r,g(r),x,g(x),'ob',lw=3)
plt.legend(['Решение с помощью\npradиальных базисных функций',
  'Исходная кривая','Данные для построения\npradиальных \
   базисных функций'])
plt.show(); plt.plot(r,-g(r)+s(r))
```

Комметрарий. По сравнению с предыдущим кодом в программу внесены некоторые изменения. Для уменьшения кода введены две функции: RBF – для задания радиальной базисной функции; g – для задания исходной функции. Изменена матрица A. Теперь она состоит из модулей разностей $\left|x_i - x_j\right|, i, j = \overline{0, n-1}$. Результаты работы программы представлены на рис. 4.15 и 4.16.

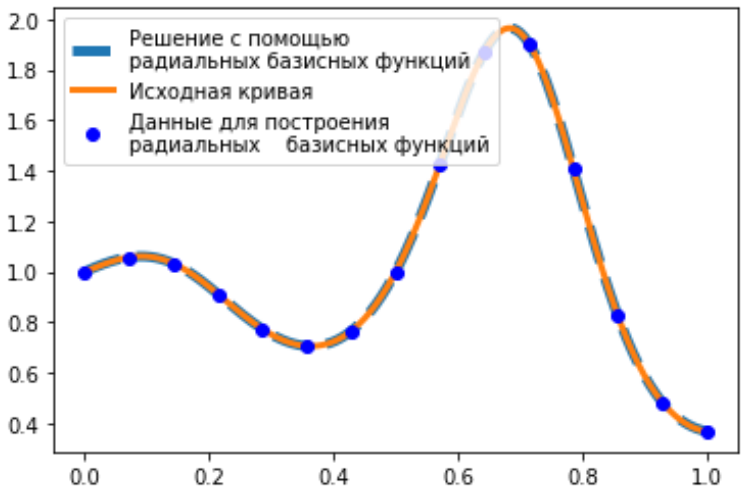

Рис. 4.15. Решение задачи интерполяции с помощью радиальной базисной функции $\varphi(r) = \dfrac{1}{1+(\varepsilon r)^2}$

Погрешность интерполяции изображена на рис. 4.16.

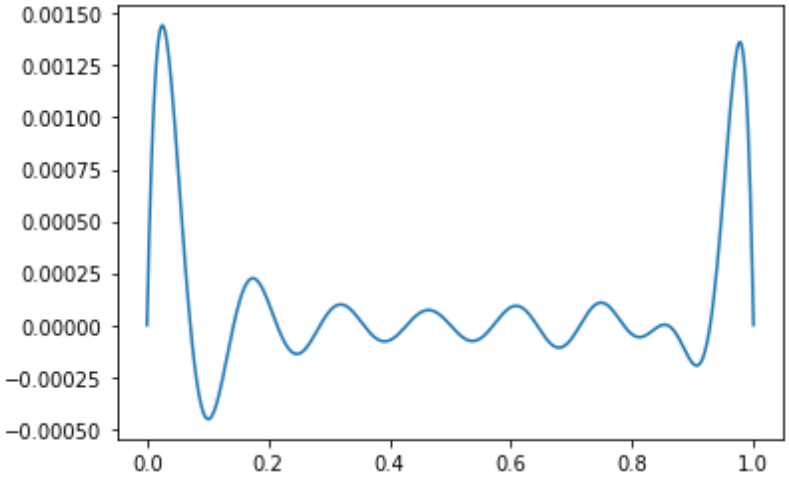

Рис. 4.16. Погрешность интерполяции

В Python для сглаживания/интерполяции рассеянных данных с помощью радиальных базисных функций предназначен класс Rbf. Обязательными параметрами класса являются исходные данные *x* и *y*. По умолчанию в качестве нормы используется евклидово расстояние, а в качестве радиальных базисных функций (параметр kind) – функции 'multiquadric' $\left(\varphi(r)=\sqrt{1+(\varepsilon r)^2}\right)$. Значение ε можно задать через параметр epsilon (epsilon=1/ε). Другими значениями параметра kind являются {'inverse', 'gaussian', 'linear', 'cubic', 'quintic', 'thin_plate'}.

Визуально различия при применении различных радиальных базисных функций для решения задачи интерполяции по одним и тем же исходным данным при небольшом количестве узлов интерполяции можно увидеть на рис. 4.17. Код программы для его получения:

```
from scipy import interpolate;from scipy.interpolate import Rbf
import matplotlib.pyplot as plt
from numpy import *;import numpy as np
def f(x): # Задаем исходную функцию
  return np.exp(x*np.cos(3*np.pi*x))
n=5; # Задаем количество узлов интерполяции
x = np.linspace(0,1,n) # Задаем интервал интерполяции
# Задаем количество точек для построения графиков
r=linspace(0,1,150)
yr=f(r); # Вычисляем значение функции в узловых точках
# Строим графики исходной кривой ("True"), наносим на
# график точки для построения RBF и сами кривые с указанием
```

```
# вида используемых радиальных базисных функций
plt.plot(r,yr,label="True")
plt.plot(x,f(x),'ok',ms=8,label='Data')
for kind in ['multiquadric','inverse','gaussian',
             'linear','cubic','quintic','thin_plate']:
  rbf = Rbf(x, f(x),function=kind)
  yy1=rbf(r);  plt.plot(r,yy1,lw=2,label=kind)
plt.legend();plt.show()
```

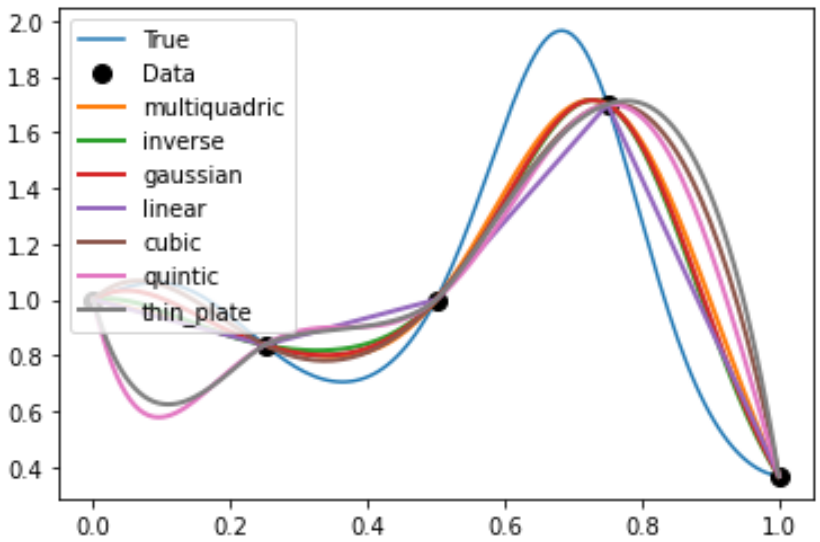

Рис. 4.17. Решение задачи интерполяции при различных значениях параметра kind

Можно самим задать нужную радиальную базисную функцию с конкретным значением параметра ε следующим образом:

```
# Задаем RBF Гаусса с параметром ε=3
def RBF(t):
  h=np.exp(-(3*t)**2)
  return h
# Передача RBF Гаусса в качестве параметра метода
rbf = Rbf(x, f(x), function=RBF)
```

Для того чтобы проиллюстрировать различие в решениях задачи интерполяции с помощью классов UnivariateSpline и Rbf, найдем решение примера 4.15 двумя указанными способами:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import Rbf, InterpolatedUnivariateSpline
```

```
x = np.array([1,2,4,7]);y = np.array([9,12,6,9])
s = interpolate.InterpolatedUnivariateSpline(x, y)
rbf = Rbf(x, y);xnew = np.arange(0, 7.1,.1);ynew = s(xnew)
xx=[0,6];yy=s(xx);yy1=rbf(xx)
plt.plot(xx,yy,'*k',ms=12);plt.plot(xx,yy1,'*b',ms=12)
plt.legend(['Решение задачи методом\nInterpolatedUnivariateSpline',
        'Решение с помощью\nрадиальных базисных функций'])
plt.plot(x, y, 'x', xnew, ynew,xnew,rbf(xnew))
plt.grid(); plt.show()
```

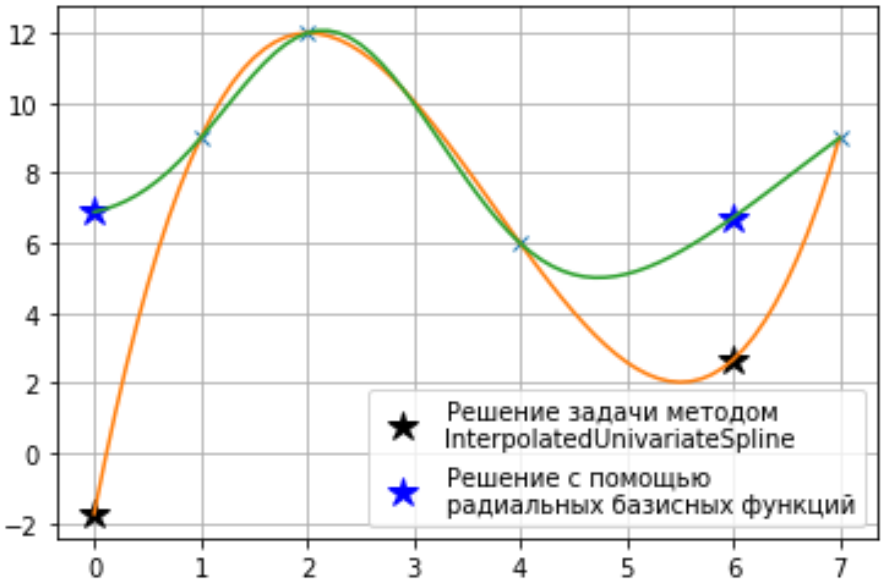

Рис. 4.18. Два способа решения задачи интерполяции

Из графиков на рис. 4.18 видно, что «сглаженная» кривая меньше подвержена осцилляции, чем сплайн, построенный по тем же данным.

Приведенные функции следует использовать с осторожностью для экстраполяции за пределы заданного диапазона данных. Более подробные сведения о радиальных базисных функциях можно найти в лекции по радиальным нейронным сетям[2].

[2] Радиальные нейронные сети: сайт. URL: https://www.sgu.ru/sites/default/files/textdocsfiles/2013/12/10/lekciya_11.pdf.

Задания для самостоятельной работы

Задание 1. Найти уравнение полинома, проходящего через точки таблицы

x	−1	1	3	μ+4
y	8	6+θ	6	ν

и вычислить его значения в точках $x^* = -2$ и $x^* = 0.5$.

Задание 2. Подобрать коэффициенты зависимости $y = c_1 \ln x + c_2 x e^{-0.5x} + c_3 x^2$ таким образом, чтобы полученная кривая прошла через точки таблицы

x	2	3	ν+5
y	4	μ	24

Задание 3. Найти уравнение двух кубических сплайнов, проходящих через точки таблицы

x	1	2	4
y	4	ν	25+μ

Значение вторых производных от каждого сплайна в первой и последней точках таблицы положить равными нулю.

Задание 4. Функция задана в виде таблицы

x	1	2+μ/(ν+1)	6	7
y	−1	6	18	μ

Вычислить значение функции в точках $x^* = 3.4$ и $x^* = 4.4$. Решить задачу интерполяции с помощью функции interp1d(), используя квадратичную и кубическую интерполяцию. Дать графическое решение задачи.

Задание 5. По данным таблицы построить графики *B*-сплайнов 2-й и 3-й степени на отрезке [1, 5]:

x	1	2+ν/10	3	4	5
y	−2	1	μ	9	ν

Задание 6. Найти уравнения кубических *B*-сплайнов, полученных с помощью функции splrep() для таблицы

x	1	2	4	5	7
y	0	1	15+ν	μ	8

Вычислить значения сплайнов в точках $x^* = 1.5$ и $x^* = 8.5$. Дать графическое решение задачи. Вычислить значения первой и второй производных в точках x^*.

Задание 7. Найти корни *B*-сплайна 3-й степени, построенного по таблице

x	1	2	3	5
y	−6	μ	ν	−6

Задание 8. Решить задачу интерполяции для таблицы

x	1	2	4	5	7
y	0	1	15+ν	μ	8

с помощью радиальной базисной функции. Вычислить значение функции в точке $x^* = 1.5$. В качестве RBF выбрать функцию $\left(\varphi(r) = \sqrt{1 + (\varepsilon r)^2}\right)$. Параметр ε взять равным μ+2.

5. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

Численное интегрирование – это вычисление определенных интегралов от функций, заданных либо в явном виде (например, $\int_0^1 e^{-\frac{x^2}{2}} dx$), либо в виде таблицы. К численному интегрированию прибегают тогда, когда интеграл невозможно вычислить точными методами либо это сделать достаточно сложно. Например, аналитическое представление подынтегральной функции известно: $\left(f(x) = e^{-\frac{x^2}{2}} \right)$, но ее первообразная не выражается через аналитические функции.

Если подынтегральная функция задана аналитически, то ее приводят к табличному виду, задавая шаг h и вычисляя значения функции в определенных точках. Интегралы вычисляют с помощью так называемых квадратурных формул, т. е. $\int_a^b f(x)dx \approx \sum_{i=0}^{n} A_i f(x_i)$. По-разному задавая коэффициенты A_i в этой формуле, получают разные формулы численного интегрирования.

SciPy имеет ряд процедур для вычисления определенных интегралов. Большинство из них находится в одном модуле scipy.integrate.

5.1. Вычисление интегралов от таблично заданных функций

Для вычисления интегралов от функций, заданных в виде таблицы в SciPy предназначены функции trapz(), cumtrapz() и simps().

Функция trapz. Вычисляет определенный интеграл по обобщенной формуле трапеций. Подынтегральная функция задана в виде таблицы

x	x_0	x_1	x_2	…	x_n
f(x)	$f(x_0)$	$f(x_1)$	$f(x_2)$	…	$f(x_n)$

Обобщенная формула трапеций при постоянном шаге имеет вид $h\left[\frac{f(x_0)+f(x_n)}{2}+\sum_{i=1}^{n-1} f(x_i)\right]$, где h – шаг таблицы $\left(h = x_{i+1} - x_i = const\right)$, а $f(x_i)$ – заданные значения функции. Если подынтегральная функция задана в явном виде и мы хотим вычислить $\int_a^b f(x)dx$ по обобщенной формуле трапеций, то функция приводится к табличному виду. Для этого интервал интегрирования разбивается на n равных (для простоты) интервалов точками x_i, $(x_i = a + i \cdot h, \text{где } h = \frac{b-a}{n}, i = \overline{0,n})$ и вычисляются значения подынтегральной функции в этих точках.

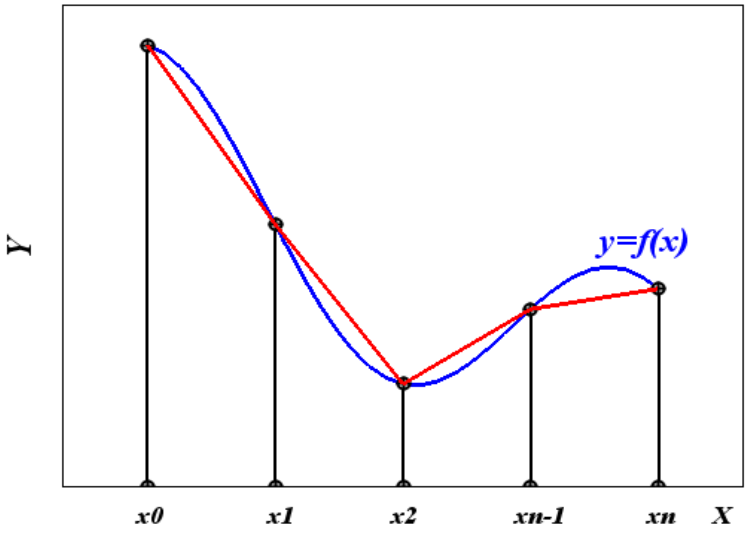

Рис. 5.1. Геометрическая интерпретация обобщенной формулы трапеций

Пример 5.1. Вычислить определенный интеграл $\int_1^2 x^2 \sin x dx$ по обобщенной формуле трапеций. Интервал интегрирования разбить на пять подынтервалов одинаковой длины.

Решение. $x_i = 1 + i \cdot h, \text{где } h = \frac{2-1}{5} = 0.2, i = \overline{0,5}.$ Составим таблицу

x	1	1,2	1,4	1,6	1,8	2
$f(x) = x^2 \sin x$	0,841471	1,342136	1,931481	2,558908	3,155266	3,63719

$$\int_1^2 x^2 \sin x dx = 0.2\left[\frac{0,841471+3,63719}{2}+1,342136+1,931481+2,558908+3,155266\right] = 2,24542457.$$

В SciPy для вычисления интегралов от табличных функций по обобщенной формуле трапеций используют функцию trapz(y):

trapz(y, x=None, dx=1.0, axis=-1)

Параметры функции trapz():

y – массив или список, состоящий из значений функции;

x – необязательный параметр. Массив или список, содержащий значения аргументов, соответствующих приведенным значениям функции. Если значение x=None, предполагается, что шаг постоянный и равен третьему параметру функции – dx;

dx – скаляр, необязательный параметр. Шаг таблицы. По умолчанию равен единице;

axis – целое число, ось (переменная), по которой производится интегрирование.

Результатом работы функции является число с плавающей точкой, если *y* – одномерный массив или массив, если размерность *y* больше единицы.

Пример 5.2. По данным примера 5.1 вычислить определенный интеграл с помощью функции trapz().

Решение. Код программы вычисления интеграла:

```
import scipy.integrate; from numpy import *
x=np.linspace(1,2,6) # задаем начальное и конечное значение x и шаг
y=[0.841471,1.342136,1.931481,2.558908,3.155266,3.63719];
a= scipy.integrate.trapz(y, x)
print("Интеграл равен %.5f" % a)
```

Результат:

Интеграл равен 2.24542.

Если подынтегральная функция задана аналитически, ее можно задать не списком значений, а формулой y=x**2*sin(x).

Функция cumtrapz(). Данная фунция аналогична функции trapz() с той лишь разницей, что выдает накопленную сумму площадей трапеций: сначала результат равен $\frac{f(x_0)+f(x_1)}{2}\cdot h,$ затем к результату добавляется $\frac{f(x_1)+f(x_2)}{2}\cdot h$ и т. д.

Приведем решение примера 5.2 с помощью функции cumtrapz():

import scipy.integrate;import numpy as np;from numpy import * x=np.linspace(1,2,6); y=x**2*sin(x)
a= scipy.integrate.cumtrapz(y, x), a

Результат:

array([0.21836073, 0.5457225 , 0.99476149, 1.56617897, 2.24542457])

Последнее значение массива – искомое значение интеграла.

Оценка погрешности обобщенной формулы трапеций: $\frac{M_2 h^2 (b-a)}{12}$. Здесь $M_2 = \max_{x\in[a,b]} |f''(x)|$.

Функция simps(). Функция simps() вычисляет определенный интеграл от функции, заданной таблицей, по обобщенной *формуле Симпсона*. Квадратурная формула Симпсона при постоянном шаге имеет вид

$$\frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right]$$

где h – шаг таблицы $\left(h = x_{i+1} - x_i = const\right)$, а $f(x_i)$ – заданные значения функции. Если подынтегральная функция задана в явном виде и мы хотим вычислить $\int_a^b f(x)dx$ по обобщенной формуле Симпсона, то интервал интегрирования разбивается на n равных (для простоты) интервалов точками x_i, $x_i = a + i \cdot h$, где $h = \frac{b-a}{n}$, $i = \overline{0,n}$. Вычисляются значения подынтегральной функции в этих точках. Число точек в формуле Симпсона должно быть нечетным.

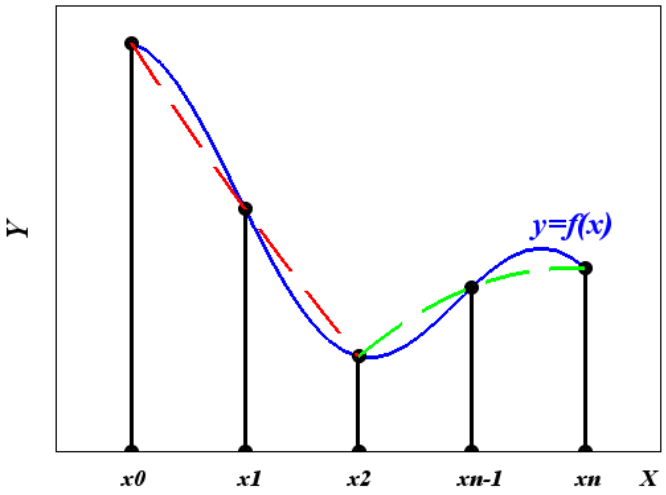

Рис. 5.2. Геометрическая интерпретация обобщенной формулы Симпсона

Оценка погрешности обобщенной формулы Симпсона: $\frac{M_4 h^4 (b-a)}{180} \cdot$ Здесь $M_4 = \max_{x\in[a,b]} \left| f^{(IV)}(x) \right| \cdot$ Применение формулы Симпсона рассмотрим на примере.

Пример 5.3. Вычислить определенный интеграл $\int_1^2 x^2 \sin x dx$ по обобщенной формуле Симпсона. Интервал интегрирования разбить на 6 подынтервалов одинаковой длины.

Решение. $x_i = 1 + i \cdot h$, где $h = \frac{2-1}{6} = \frac{1}{6}$, $i = \overline{0,6}$. Составим таблицу

x	1	1,166667	1,333333	1,5	1,666667	1,833333	2
$f(x) = x^2 \sin x$	0,841471	1,251467	1,72789	2,244364	2,765022	3,245941	3,63719

В соответствии с формулой Симпсона имеем (0.841471 + + 4·1.251467 + 2·1.72789+ 4·2.244364 + 2·2.765022 +4·3.245941 + + 3.63719)·(1/18)= 2.246198444.

Для вычисления интеграла по обобщенной формуле Симпсона используют функцию simps(). Ее синтаксис:

simps(y, x=None, dx=1, axis=-1, even='avg').

Назначение первых четырех параметров функции simps() идентично аналогичным параметрам функции trapz(). Последний параметр even отвечает за способ вычисления интеграла, если количество значений функции – четное. Значение параметра – символьная строка. Возможные значения параметра: {'avg', 'first', 'last'}. Если значение even установлено как 'avg' (average), то выдается среднее двух результатов: первый предполагает вычисления интеграла по формуле Симпсона с использованием всех точек таблицы, кроме последней, и применение формулы трапеции на интервале $[x_{n-1}, x_n]$; второй – вычисления по формуле трапеций на интервале $[x_0, x_1]$ и по формуле Симпсона на интервале $[x_1, x_n]$. При значениях параметра, равных 'first' или 'last', выдается один из этих результатов в соответствии со значением параметра even: на первых ('first') или последних ('last') интервалах применяется обобщенная формула Симпсона. Такая информация приведена в справке по функции. На самом деле все ответы выдаются одинаковыми.

Решим пример 5.3 с помощью функции simps().

Код программы:

```
import scipy.integrate
import numpy as np;from numpy import *
```

```
x=np.linspace(1,2,6)
y=x**2*sin(x)
a= scipy.integrate.simps(y, x)
a
```

Результат: 2.2457369349535465. В коде значение функции в заданных точках приведено в аналитическом виде. Можно было задать значения функции списком или массивом.

5.2. Вычисление интегралов от функций, заданных в явном виде

Для вычисления определенных интегралов от явно заданных функций в SciPy используют функции quad(), fixed_quad(), romberg().

Функция quad. Вычисляет $\int_a^b f(x)dx$ с помощью метода из библиотеки Fortran QUADPAC [1]. Синтаксис функции:

quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08, limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50).

Назначение некоторых входных аргументов функции:

func – подынтегральная функция;

a и *b* – соответственно нижний и верхний пределы интегрирования. Если нижний предел интегрирования равен -∞, в качестве аргумента *a* используют -numpy.inf. Соответственно, numpy.inf используют для предела ∞;

args – кортеж, содержащий необязательные дополнительные аргументы для передачи в подынтегральную функцию *func*. Например, для вычисления интеграла $\int_1^3 ax^2 dx$ можно передать в функцию *func* значение параметра a=3 так: args(3,);

full_output – параметр типа Int. В случае ненулевого значения выдается дополнительная информация о ходе решения задачи;

epsabs – параметр типа Float или Int. Абсолютная погрешность;

epsrel – параметр типа Float или Int. Относительная погрешность;

limit – параметр типа Float или Int. Верхняя граница числа подынтервалов, используемых в адаптивном алгоритме.

Выходными параметрами функции являются значение интеграла, оценка абсолютной погрешности результата и словарь с некоторыми дополнительными сведениями. Последние выводятся при значении параметра *full_output*, отличном от нуля.

Пример 5.4. Вычислить определенный интеграл $\int_1^3 \frac{\sin x}{x+1} dx$.

Решение:

```
import scipy.integrate
from numpy import *
f= lambda x:sin(x)/(x+1)
i = scipy.integrate.quad(f, 1, 3)
print("Интеграл равен %.3f" % i[0])
print("Оценка абсолютной погрешности результата %.12f" % i[1])
```

Результат:

```
Интеграл равен 0.557.
Оценка абсолютной погрешности результата 0.000000000000.
```

Аналогичные функции dblquad, tplquad и nquad применяют для вычисления двойных, тройных и кратных интегралов.

Пример 5.5. Вычислить $\int_1^2 \int_1^x xydxdy$.

Решение:

```
from scipy import integrate
f = lambda y, x: x*y
a,b=integrate.dblquad(f, 1, 2, lambda x: 1, lambda x: x)
print("Интеграл равен %.3f" % a)
print("Оценка абсолютной погрешности результата %.12f" % b)
```

Результаты:

```
Интеграл равен 1.125.
Оценка абсолютной погрешности результата 0.000000000000.
```

Альтернативный вариант решения:

```
from scipy import integrate
def f(x,y):
```

```
  return x*y
def p(x):
  return x
a,b=integrate.dblquad(f, 1, 2, lambda x: 1, p)
print("Интеграл равен %.3f" % a)
print("Оценка абсолютной погрешности результата %.12f" % b)
```

Функция fixed_quad. Вычисляет $\int_a^b f(x)dx$ с помощью квадратурной формулы Гаусса. В квадратурной формуле Гаусса узлы и весовые коэффициенты подбираются так, чтобы при заданном числе узлов она являлась точной для многочленов максимально высокой степени ($\leq 2n$-1).

Для этого производят следующие преобразования. Делают замену переменных:

$$x = \frac{a+b}{2} + \frac{b-a}{2}t.$$

Тогда

$$\int_a^b f(x)dx = \frac{b-2}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}t\right)dt.$$

Применение к последнему интегралу квадратурной формулы Гаусса дает

$$\frac{b-2}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}t\right)dt = \frac{b-a}{2}\sum_{i=1}^{n} A_i f\left(\frac{a+b}{2} + \frac{b-a}{2}t_i\right),$$

где t_i – нули полинома Лежандра, а A_i – решение системы линейных уравнений:

$$\begin{cases} \sum_{i=1}^{n} A_i = 2 \\ \sum_{i=1}^{n} A_i t_i = 0 \\ \sum_{i=1}^{n} A_i t_i^2 = \frac{2}{3} \\ \sum_{i=1}^{n} A_i t_i^3 = 0 \\ \ldots\ldots\ldots\ldots\ldots \\ \sum_{i=1}^{n} A_i t^k = \frac{1-(-1)^{k+1}}{k+1} \end{cases}.$$

Покажем на примере, как в квадратурной формуле Гаусса находят коэффициенты A_i и точки, в которых нужно вычислить значения подынтегральной функции.

Пример 5.6. Вычислить $\int_1^2 x^2 \sin x dx$ по квадратурной формуле Гаусса с использованием трех узлов.

Решение. Приводим пределы интегрирования к интервалу [−1, 1]:

$$\int_1^2 x^2 \sin x dx = \frac{2-1}{2} \int_{-1}^1 (1.5+.5t)^2 \sin(1.5+.5t) dt.$$

Уравнение полинома Лежандра 3-й степени найдем с помощью соответствующей функции библиотеки SymPy. Для наглядности построим график полинома Лежандра 3-й степени:

```
from sympy import *;t=symbols('t');y1=legendre_poly(3,t)
p='Уравнение полинома\n  '
h=latex(y1,mode='inline',fold_short_frac=False)
plot(y1,(t,-1,1),line_color='r',label=p+h,legend=True)
```

Краткий комментарий к программе. Объявляем *t* символьной переменной. Находим полином Лежандра 3-й степени (*y*1). Готовим к выводу на экран текст легенды с уравнением полинома стилем LaTex. Выводим на график легенду.

Выполнив программу, получаем график (рис. 5.3).

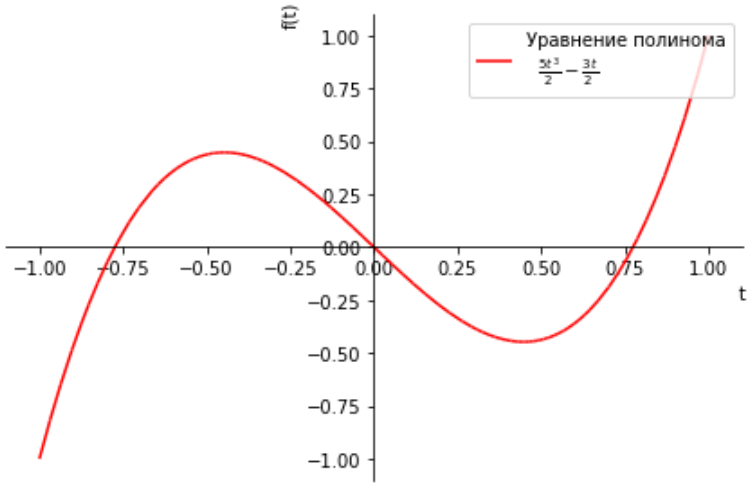

Рис. 5.3. График полинома Лежандра 3-й степени

Из рис. 5.3 видно, что полином имеет три корня. Для полинома Лежандра 3-й степени их найти несложно: полином имеет вид $\frac{5t^3}{2}-\frac{3t}{2}$.

$\frac{5t^3}{2}-\frac{3t}{2}=0, t(5t^2-3)=0, t_1=0, t_{2,3}=\pm\sqrt{\frac{3}{5}}.$ Далее рассчитываем точки $\left(\frac{a+b}{2}+\frac{b-a}{2}t_i\right)$, в которых будет вычисляться подынтегральная функция (a = 1, b = 2). Эти точки: [1.11270167, 1.5, 1.88729833]. Вычисляем значения подынтегральной функции $y=x^2\sin x$ в найденных точках: [1.11045234, 2.24436372, 3.38497554].

Составляем систему линейных уравнений для нахождения трех (в нашем примере) коэффициентов $A_i,\ i=1,2,3$:

$$\begin{cases} A_1+A_2+A_3=2 \\ -\sqrt{\frac{3}{5}}A_1+\sqrt{\frac{3}{5}}A_3=0. \\ \frac{3}{5}A_1+\frac{3}{5}A_3=\frac{2}{3} \end{cases}$$

Решая ее, получаем $A_1=A_3=\frac{5}{9}, A_2=\frac{8}{9}$.

Подставляя полученные значения в итоговую формулу, имеем

$$\int_1^2 x^2\sin x dx\approx\frac{2-1}{2}\left(1.11045234\cdot\frac{5}{9}+2.24436372\cdot\frac{8}{9}+3.38497554\cdot\frac{5}{9}\right)\approx 2.24622495.$$

Последние вычисления можно провести в Python:

```
import numpy as np
from numpy import *
a,b=1,2
A=np.array([5/9,8/9,5/9])
x=np.array([-(3/5)**.5,0,(3/5)**.5])
t=(a+b)/2+(b-a)/2*x
y=t**2*np.sin(t);sum(y*A)*(b-a)/2
```

Результат:

2.2462249513719477.

На рис. 5.4 приведено графическое решения задачи.

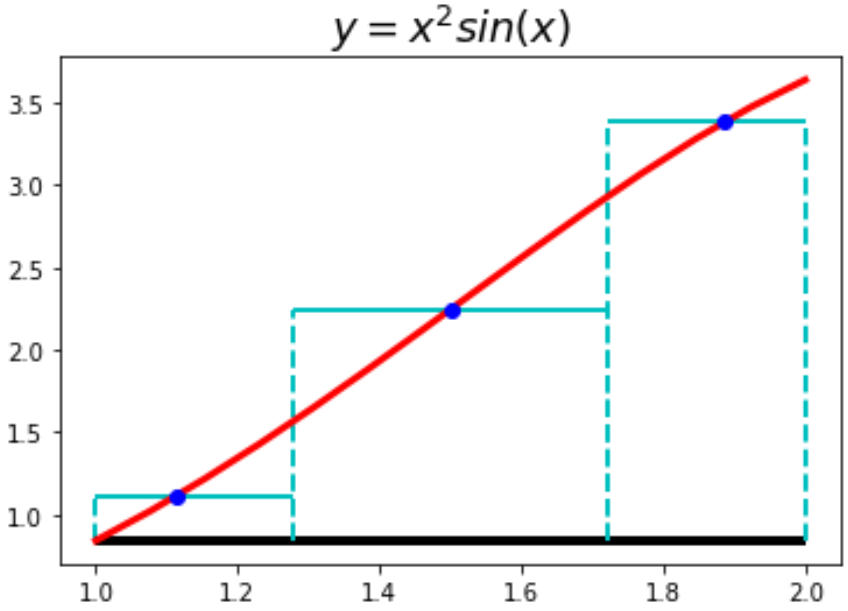

Рис. 5.4. Геометрическая интерпретация квадратурной формулы Гаусса (три узла)

Высоты прямоугольников равны $f\left(\frac{a+b}{2}+\frac{b-a}{2}t_i\right)$, основания – $\frac{b-a}{2}\cdot A_i, \quad i=0,1,2.$

При поиске коэффициентов A_k можно обойтись без решения системы уравнений, а вычислить их по формуле $A_k=\frac{2}{(1-t_k^2)\left[P_n'(t_k)\right]^2}$, где $P_n(t)$ – полином Лежандра степени n, а t_k – его корни. С учетом возможностей библиотеки SymPy коэффициенты могут быть найдены таким образом:

```
import numpy as np
from numpy import *
from sympy import *
t=symbols('t')
x=np.array([-(3/5)**.5,0,(3/5)**.5])
for i in range(len(x)):
  y1=legendre_poly(3,t)
  w=y1.diff(t).subs(t,x[i])
  A=2/(1-x[i]**2)/w**2
  print("Коэффициент A(%i)=%.7f" % (i+1,A))
```

Выполнив программу, получаем:

```
Коэффициент A(1)=0.5555556.
Коэффициент A(2)=0.8888889.
Коэффициент A(3)=0.5555556.
```

В приведенном коде *w* – значения производной полинома Лежандра в точках x_i.

Конечно, можно не искать значения t_i и A_k, а взять их из соответствующих таблиц.

От всех этих громоздких вычислений нас избавляет использование функции fixed_quad() библиотеки SciPy.

Синтаксис функции: fixed_quad(func, a, b, args=(), n=5).

Входные параметры функции:

func – подынтегральная функция;

a и *b* – соответственно нижний и верхний пределы интегрирования;

args – кортеж, содержащий необязательные дополнительные аргументы для передачи в подынтегральную функцию *func*;

n – порядок квадратуры метода Гаусса.

Выходным параметром является кортеж, содержащий значение интеграла и значение None. Рассмотрим использование этой функции на примере.

Пример 5.7. С помощью квадратурной формулы метода Гаусса вычислить определенный интеграл $\int_1^2 x^2 \sin x dx$. Порядок квадратурной формулы взять равным 3 и 5. Полученные результаты сравнить с точным значением интеграла. Оценить абсолютную погрешность результатов.

Решение:

```
import scipy.integrate
from numpy import *
f= lambda x:x**2*sin(x)
a= scipy.integrate.fixed_quad(f, 1, 2,n=3)
print("Интеграл равен %.10f" % a[0])
```

Результат:

```
Интеграл равен 2.24622495.
```

Если изменить порядок квадратурной формулы с 3 до 5 (по умолчанию), получим значение 2.246239. Точное значение интеграла $\int_1^2 x^2 \sin x dx$ можно вычислить с помощью функции Integral() библиотеки SymPy:

```
from sympy import *
from sympy.abc import *
y=x**2*sin(x)
a=Integral(y,x)
z=a.doit()
p=(z.subs(x,2)-z.subs(x,1)).n()
```

Если вывести на экран значение z, получим $-x^2\cos(x)+2x\sin(x)+2\cos(x).$

Вычисление определенного интеграла (переменная p) дает 2.24623910491308. Абсолютная погрешность в первом случае (n = 3) равна $1.415354\cdot10^{-5}$, во втором (n = 5) – $3.509948\cdot10^{-11}$.

Квадратурная формула Гаусса обеспечивает очень высокую точность при небольшом числе узлов. Более строгое значение оценки погрешности квадратурных формул Гаусса при различных значениях *n* можно найти в учебном пособии Р. З. Даутова и М. Р. Тимербаева[3].

Результаты использования этой функции показали, что значения интегралов, вычисленные с помощью функций quad() и fixed_quad(), практически совпадают (в рассмотренных примерах – до 10 знаков после запятой), если значение *n* в функции fixed_quad() равно 10.

Функция romberg(). Вычисляет $\int_a^b f(x)dx$ с помощью метода Ромберга, который заключается в последовательном уточнении значения интеграла при кратном увеличении числа разбиений.

Синтаксис функции romberg():

romberg(function, a, b, args=(), tol=1.48e-08, rtol=1.48e-08, show=False, divmax=10, vec_func=False).

Входные параметры функции:

function – подынтегральная функция;

[3] Даутов Р. З., Тимербаев М. Р. Численные методы. Приближение функций: учеб. пособие. Казань: Изд-во Казан. ун-та, 2021. 123 с.

a и *b* – пределы интегрирования;

args – кортеж, содержащий необязательные дополнительные аргументы для передачи в подынтегральную функцию *func*. По умолчанию никакие аргументы в функцию не передаются;

tol, rtol – желаемая абсолютная и относительная погрешности;

show – переменная типа bool. Если значение параметра *show* равно 1 (или True), выдаются промежуточные результаты расчетов по методу Ромберга;

vec_func – переменная типа bool. Если значение параметра *vec_func* равно True (по умолчанию False), предполагается, что функция задает вектор аргументов (указывает, обрабатывает ли *func* массивы в качестве аргументов, т. е. является ли она «векторной» функцией);

divmax – переменная типа int. Максимальный порядок экстраполяции. Значение по умолчанию равно 10.

Решение примера 5.7 с помощью функции romberg():

```
import scipy.integrate
import numpy as np
from numpy import *
f= lambda x:x**2*sin(x)
scipy.integrate.romberg(f,1,2,show=True)
```

Результат:

```
Romberg integration of <function vectorize1.<locals>.vfunc at 0x7fc50e510950> from [1, 2]

 Steps  StepSize  Results
     1  1.000000  2.239330
     2  0.500000  2.241847  2.242686
     4  0.250000  2.244985  2.246031  2.246254
     8  0.125000  2.245916  2.246226  2.246239  2.246239
    16  0.062500  2.246158  2.246238  2.246239  2.246239  2.246239

The final result is 2.2462391049148427 after 17 function evaluations.
2.2462391049148427
```

Поясним полученные результаты. Первое значение (2.239330) было получено в результате применения формулы трапеции к вычислению интеграла $\int_1^2 x^2 \sin x dx$. Это значение равно

$$\frac{f(a)+f(b)}{2}\cdot\frac{(b-a)}{2^0}=\frac{f(1)+f(2)}{2}(2-1),$$

где $f(x)=x^2\sin x.$ Число разбиений отрезка [a, b] n = 0 (отрезок не разбивался). Затем обозначили результат вычислений R(1,1). Интервал интегрирования разбивался на два отрезка одинаковой длины (n = 1). Вычисляли интеграл по обобщенной формуле трапеций: $R(2,1)=h\left(\frac{f(1)}{2}+f\left(\frac{1+2}{2}\right)+\frac{f(2)}{2}\right)$, где $h=\frac{2-1}{2^n}=\frac{2-1}{2^1}=0.5.$ Результат равен 2.241847. Далее вычислялась величина |R(2, 1)-R(1, 1)|. Поскольку полученное значение больше установленной по умолчанию погрешности *tol*, результат корректировался. Вычисляли R(2, 2) по формуле

$$R(2,2)=R(2,1)+\frac{R(2,1)-R(1,1)}{2^{2n}-1}=2.241847+\frac{2.241847-2.239330}{2^2-1}=2.242686.$$

|R(2, 2)-R(1, 1)|>tol, поэтому интервал интегрирования разбивался на 2^n=4 части (n = 2) и вычислялся искомый интеграл по обобщенной формуле трапеций уже по 5 точкам: $R(3,1)=h\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{3}f(a+ih)\right)$, где $h=\frac{2-1}{2^2}=0.25.$ R(3,1)=2.244985. Поскольку требуемая точность не достигнута, вычислялось значение R(3, 2) и т. д. Получилась треугольная таблица:

R(1, 1)
R(2, 1) R(2, 2)
R(3, 1) R(3, 2) R(3, 3)
R(4, 1) R(4, 2) R(4, 3) R(4, 4)
R(5, 1) R(5, 2) R(5, 3) R(5, 4) R(5, 5)

При вычислении каждого последующего значения R(k, 1) происходило удвоение числа отрезков, на которые разбивался интервал интегрирования. Уточненные значения других элементов таблицы вычислялись по рекуррентной формуле

$$R(n+1,m+1)=R(n+1,m)+\frac{R(n+1,m)-R(n,m)}{2^{2m}-1}.$$

Погрешность метода: $O(h_n^{2m+2}),$ где $h_n=\frac{b-a}{2^n}.$

Задания для самостоятельной работы

Задание 1. Взяв шаг, равный одной десятой длины интервала интегрирования, вычислить определенный интеграл по обобщенным формулам трапеций и Симпсона:

а) $\int\limits_{\nu+0.2\mu-1}^{2+1.3\nu} e^{\frac{(x-\nu)^2}{8}}\,dx$; б) $\int\limits_{\nu+0.1\mu}^{5+1.3\nu} \sqrt{x^2+\nu+\gamma}\,dx$.

Тот же интеграл вычислить средствами Python с помощью одной из приведенных в пособии специальных функций.

Задание 2. Составить программу для вычисления интеграла из задания 1 по обобщенным формулам прямоугольников, трапеций и Симпсона.2

6. ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ И СИСТЕМ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

Обыкновенным дифференциальным уравнением (ОДУ) порядка n называется уравнение вида $F(x, y(x), y'(x), y''(x),...,y^{(n)}(x)) = 0$, где n – порядок наивысшей производной, входящей в уравнение. В каноническом виде ОДУ n-го порядка записывается как

$$y^{(n)}(x) = f(x, y(x), y'(x), y''(x),...,y^{(n-1)}(x)).$$

Под интегрированием дифуравнения понимают нахождение функции $y(x)$, которая удовлетворяет этому уравнению. $y(x)$ называется решением дифференциального уравнения. Общее решение ОДУ n-го порядка имеет вид $y = y(x, c_1, c_2,...,c_n)$, где c_1, c_2, … , c_n – произвольные константы.

Системой обыкновенных дифференциальных уравнений первого порядка называется система уравнений вида

$$\begin{cases} \dfrac{dy_1(x)}{dx} = f_1(x, y_1(x), y_2(x),...,y_n(x)) \\ \dfrac{dy_2(x)}{dx} = f_2(x, y_1(x), y_2(x),...,y_n(x)) \\ ..... \\ \dfrac{dy_n(x)}{dx} = f_n(x, y_1(x), y_2(x),...,y_n(x)) \end{cases}.$$

Здесь x – независимая переменная; f_1, f_2, …, f_n – заданные функции. Решением этой системы называют совокупность функций $y_1(x) = \varphi_1(x),...,y_n(x) = \varphi_n(x)$, которые после подстановки в систему уравнений обращают их в тождества.

Дифференциальные уравнения и системы дифференциальных уравнений имеют бесчисленное множество решений, отличающихся друг от друга набором констант. Для того чтобы выделить среди бесчисленного множества решений одно решение, необходимо задать начальные и/или граничные условия. Количество таких условий должно совпадать с порядком дифференциального уравнения или с количеством дифференциальных уравнений в системе дифуравнений.

В зависимости от вида этих дополнительных условий различают *задачу Коши* и *краевую задачу*.

Задача Коши для ОДУ первого порядка ставится следующим образом: найти решение дифференциального уравнения $y' = f(x, y)$, проходящее через точку с координатами (x_0, y_0), где x_0 и y_0 – некоторые заданные числа:

$$\begin{cases} y' = f(x, y) \\ y(x_0) = y_0 \end{cases}.$$

Задача Коши для дифференциального уравнения порядка n есть задача о нахождении частного решения уравнения, удовлетворяющего начальным условиям $y(x_0) = y_0, y'(x_0) = y'_0, ..., y^{(n-1)}(x_0) = y_0^{(n-1)}$. Здесь $y_0, y'_0, ..., y_0^{(n-1)}$ – некоторые заданные числа.

Графическое изображение частного решения называют интегральной кривой. Общее решение дифференциального уравнения n-го порядка определяет n-параметрическое семейство интегральных кривых.

6.1. Решение задачи Коши для дифференциального уравнения первого порядка

Если дифференциальное уравнение не удается решить точными методами, его решают численно. Рассмотрим численные методы решения задачи Коши для ОДУ первого порядка: $\begin{cases} y' = f(x, y) \\ y(x_0) = y_0 \end{cases}$.

Результатом численного решения дифференциального уравнения является таблица

x	x_0	x_1	x_2	…	x_n
y	y_0	y_1	y_2	…	y_n

Первая точка таблицы задана в условии задачи. Задавая шаг h_i, вычисляют значения $x_1, x_2, \dots, x_n$: $x_{i+1}=x_i+h_i$, $i=0, 1, \dots, n-1$. Шаг может быть как переменным, так и постоянным. y_i вычисляют по формулам, соответствующим методу решения. Рассмотрим некоторые из них.

Метод Эйлера. В методе Эйлера значения y_{i+1} вычисляют по формуле $y(x_{i+1}) = y(x_i) + h_i f(x_i, y_i), i = \overline{0, n-1}$. Работу метода рассмотрим на примере.

Пример 6.1. Решить задачу Коши для ОДУ первого порядка $\begin{cases} y' = \sin x + y \\ y(0) = 2 \end{cases}$ методом Эйлера. Сделать три шага (h = 0.1). Сравнить полученное решение с известным точным решением задачи $y = \frac{5}{2}e^x - \frac{\sin x}{2} - \frac{\cos x}{2}$. Результаты свести в таблицу.

Решение. В соответствии с формулой метода Эйлера $y(x_1) = y(x_0) + hf(x_0, y_0)$. Здесь x_0 = 0, y_0 = 2, h = 0.1, $f(x,y)$=sin(x)+y. Тогда $y_1 = y(x_1) = 2 + 0.1(\sin 0 + 2) = 2.2$. Находим следующую точку: x_1=x_0+h=1.1, $y(x_2) = y(x_1) + hf(x_1, y_1)$. $y_2 = y(x_2) = 2.2 + 0.1(\sin 0.1 + 2.2) = 2.429983$. Третий шаг: x_2=x_1+h=0.2, $y_3 = y(x_3) = 2.429983 + 0.1(\sin 0.2 + 2.429983) = 2.692848$. Вычисляем точное значение функции в точках 0.1, 0.2 и 0.3 и результаты сводим в табл. 6.1.

Таблица 6.1

Результаты решения ОДУ методом Эйлера

x	y (метод Эйлера)	y точное
0	2	2
0.1	2.2	2.2155085
0.2	2.429983	2.46413894
0.3	2.692848	2.74921867

Из табл. 6.1 видно, что по мере удаления от начальной точки абсолютная разность между точным и приближенным решениями возрастает. Метод Эйлера довольно простой, но не очень точный. Его погрешность $o(h)$.

Рассмотрим более точный метод решения задачи Коши для ОДУ первого порядка.

Метод Рунге–Кутта. Это метод, в котором следующее значение функции y_{i+1} вычисляется по формуле

$$y_{i+1} = y_i + \frac{k_{1i} + 2k_{2i} + 2k_{3i} + k_{4i}}{6},$$

где

$$k_{1i} = h_i f(x_i, y_i),$$

$$k_{2i} = h_i f(x_i + \frac{h_i}{2}, y_i + \frac{k_{1i}}{2}),$$

$$k_{3i} = h_i f(x_i + \frac{h_i}{2}, y_i + \frac{k_{2i}}{2}),$$

$$k_{4i} = h_i f(x_i + h, y_i + k_{3i}).$$

Решим пример 6.1 методом Рунге–Кутта.

$$y_1 = y_0 + \frac{k_{10} + 2k_{20} + 2k_{30} + k_{40}}{6},\ k_{10} = hf(x_0, y_0) = 0.1(\sin(0) + 2) = 0.2;$$

$$k_{20} = hf(x_0 + \frac{h}{2}, y_0 + \frac{k_{10}}{2}) = 0.1 f\left(0.05; 2 + \frac{0.2}{2}\right) = 0.1 f(0.05, 2.1) = 0.1(\sin 0.05 + 2.1) =$$

$$= 0.2149979;\ k_{30} = hf(x_0 + \frac{h}{2}, y_0 + \frac{k_{20}}{2}) = 0.1 f\left(0.05; 2 + \frac{0.2149979}{2}\right) =$$

$$= 0.1 f(0.05, 2.107499) = 0.1(\sin 0.05 + 2.107499) = 0.2157478;\ k_{30} = hf(x_0 + h, y_0 + k_{30}) =$$

$$= 0.1 f(0.1, 2 + 0.2157478) = 0.1(\sin(0.1) + 2.2157478) = 0.2315581.$$

$$y_1 = 2 + \frac{0.22 + 2 \cdot 0.2149979 + 2 \cdot 0.2157478 + 0.2315581}{6} = 2.2155083.$$

Аналогично можно найти y_2 и y_3: y_2=2.46413841, y_3=2.74921778. Результаты сведем в табл. 6.2.

Таблица 6.2

Результаты решения ОДУ методом Рунге–Кутта

x	y (метод Рунге-Кутта)	y точное
0	2	2
0.1	2.2155083	2.2155085
0.2	2.46413841	2.46413894
0.3	2.74921778	2.74921867

Вывод. Метод Рунге–Кутта значительно точнее метода Эйлера. Порядок точности метода Рунге–Кутта $o(h^4)$. За точность приходится платить значительно большим количеством вычислений.

Оба метода нетрудно запрограммировать. Далее приведен код решения расмотренной задачи обоими методами. Сделан 21 шаг:

```
import numpy as np; import matplotlib.pyplot as plt
def f(x,y):
```

```
  z=np.sin(x)+y
  return z
p=np.array([0,2]);a=0;b=3;n=21
s=p[1]
x=np.linspace(a,b,n)
h=(b-a)/(n-1) # шаг
# МЕТОД РУНГЕ-КУТТА
ypk=np.zeros(n);ypk[0]=p[1] # значение функции в точке x[0]
# В цикле формируем еще n-1 значение функции
for i in range(n-1):
  k1=h*f(x[i],s);k2=h*f(x[i]+h/2,s+k1/2)
  k3=h*f(x[i]+h/2,s+k2/2);k4=h*f(x[i]+h,s+k3)
  t=(k1+2*k2+2*k3+k4)/6;s+=t;# значение функции в точке x[i+1]
  ypk[i+1]=s;
# Метод Эйлера
yei=np.zeros(n)
s1=p[1]
yei[0]=p[1] # значение функции в точке x[0]
# В цикле формируем еще n-1 значение функции
for i in range(n-1):
  t=h*f(x[i],s1)
  s1+=t
  yei[i+1]=s1
yt=-np.sin(x)/2-np.cos(x)/2+5/2*np.exp(x)
print("Максимальная погрешность метода Эйлера равна %.8f"
     % np.max(abs(yt-yei)))
print("Максимальная погрешность метода Рунге-Кутта равна %.8f"
     % np.max(abs(yt-ypk)))
plt.plot(x,ypk,'--r*',x,yt,'--k',x,yei,'--g',lw=2,ms=7)
plt.legend(['Метод Рунге-Кутта','Точное решение',
          'Метод Эйлера'])
plt.grid()
plt.show()
```

Результатами работы программы являются значения максимальной по модулю погрешности обоих методов и графическое решение задачи на интервале [0, 3] с шагом 0.1:

Максимальная погрешность метода Эйлера равна 9.92519172.

Максимальная погрешность метода Рунге–Кутта равна 0.00060313.

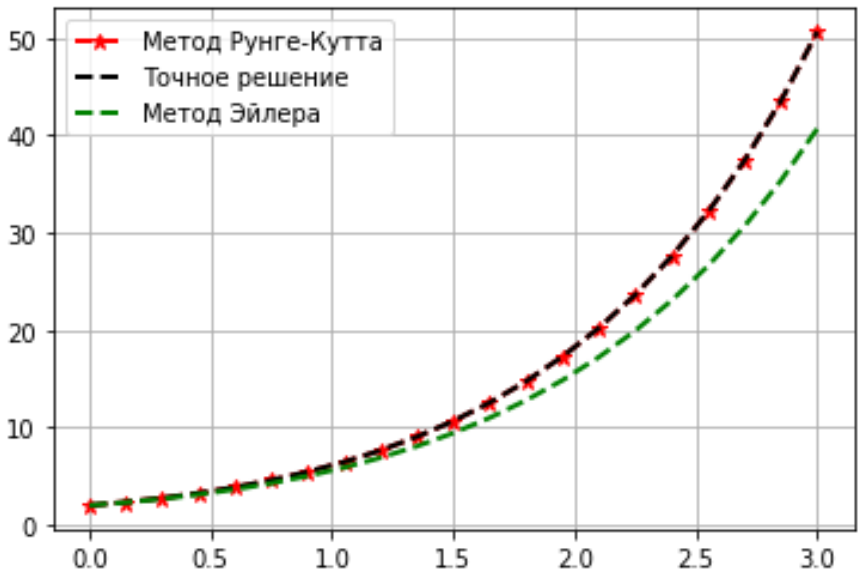

Рис. 6.1. Точное и численное решение задачи Коши для ОДУ первого порядка

Погрешность можно уменьшить, увеличив количество точек *n*.

Для решения ОДУ в библиотеке SciPy есть несколько функций. Среди них – odeodeint() и solve_ivp(). Рассмотрим первую из них.

Синтаксис функции odeint():

odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0, tfirst=False).

Некоторые параметры функции odeint(), из которых обязательными являются лишь первые три:

func – функция, задающая правую часть дифференциального уравнения $\frac{dy}{dt} = func(y,t,\ldots)$ первого порядка или правые части системы таких уравнений. Если порядок следования аргументов в функции $func(t,y,\ldots)$, значение параметра tfirst должно быть True;

y0 – вектор начальных условий;

t – последовательность. Точки интервала, на котором необходимо получить решение. Начальная точка должна быть первым элементом этой последовательности. Последовательность должна быть монотонно возрастающей или монотонно убывающей. Допускаются повторяющиеся значения;

args – кортеж, необязательные дополнительные аргументы для передачи в функцию;

Dfun – градиент функции;

col_deriv – переменная типа bool. Аргумент должен принимать значение True, если порядок производных в Dfun – по столбцам, и False – в противном случае;

full_output – переменная типа bool. Принимает значение True, если мы хотим получить словарь с дополнительными подробностями решения задачи;

rtol и *atol* – определяют относительную и абсолютную погрешности вычислений значений y_i. Могут быть векторами или скалярами. По умолчанию rtol = atoll = 1.49012e-8;

printmessg – переменная типа bool. Используется, если мы хотим вывести сообщение о сходимости процесса решения.

Рассмотрим применение этой функции для решения примера 6.2.

Пример 6.2. Решить задачу Коши для ОДУ первого порядка $\begin{cases} y' = \sin x + y \\ y(0) = 2 \end{cases}$. Решение получить в точках отрезка [0, 3] с шагом 0.01, используя функцию odeint(). Нанести на график полученное решение и известное точное решение задачи $y = \frac{5}{2}e^x - \frac{\sin x}{2} - \frac{\cos x}{2}$. Вычислить максимум модуля разности между точным и приближенным значениями в 301-й точке отрезка [0, 3].

Решение:

```
import numpy as np; import matplotlib.pyplot as plt
from scipy.integrate import odeint
def func(y,x):
   dydx=np.sin(x)+y;
   return dydx
x2 = np.linspace(0,3,301) # интервал поиска решения
y0 = [2] # Значения функции  в точке x2[0]
y = odeint(func, y0, x2);
yt=-np.sin(x2)/2-np.cos(x2)/2+5/2*np.exp(x2)
plt.plot(x2,y,':r*',x2,yt,'--g',lw=3,ms=4)
plt.legend(['Численное решение','Точное решение'])
p=np.max(abs((yt-y.reshape(1,len(x2)))))
print("Максимальная погрешность метода odeint равна %.8f" % p)
plt.show()
```

Результаты решения представлены на рис. 6.2.

Результатом работы программы является также и значение максимальной абсолютной погрешности. Максимальная погрешность метода odeint равна 0.00000368.

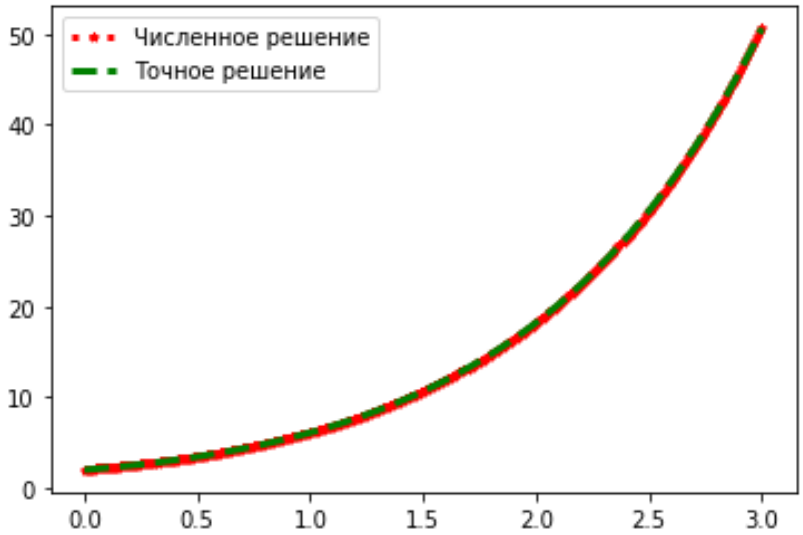

Рис. 6.2. Точное и приближенное решение задачи Коши

Оба графика практически неразличимы: точное решение задачи практически совпало с приближенным, что подтверждает и найденное значение максимума модуля разности между точным и приближенным решениями.

При необходимости получить числовые значения решения в приведенной программе нужно вывести на экран значения массива *y*.

Рассмотрим еще одну функцию для решения ОДУ – функцию solve_ivp (). Ее синтаксис:

solve_ivp (fun , t_span , y0 , method = 'RK45' , t_eval = None , visible_output = False , events = None , vectorized = False , args = None , ** options).

С помощью этой функции можно численно решить задачу Коши как для ОДУ первого порядка, так и для системы обыкновенных дифференциальных уравнений.

Назначение некоторых параметров функции:

fun – правые части системы дифуравнений. Вызываемая функция задается в виде fun(t, y);

t_span – кортеж из двух чисел, являющихся началом и концом интервала интегрирования;

y0 – массив начальных значений. Размерность равна количеству уравнений в системе;

method – метод решения задачи. Может быть одним из следующих: 'RK45' – метод Рунге–Кутта порядка 4; 'RK23' – аналогичный метод порядка 2; 'DOP853' – метод Рунге–Кутта порядка 8; 'Radau' – неявный метод Рунге–Кутта семейства Радау порядка 5; 'BDF' – неявный многошаговый метод переменного порядка (от 1 до 5), основанный на формуле обратного дифференцирования для аппроксимации производной; 'LSODA' – метод Адамса/BDF с автоматическим определением жесткости;

t_eval – точки, в которых нужно вычислить значение функции. Они должны быть отсортированы и лежать в пределах *t_span*. Если значение равно None (по умолчанию), используются точки, выбранные в соответсвии с методом решения.

С остальными параметрами можно ознакомиться в документации к функции.

Среди выходных параметров основными являются:

t – точки, в которых получено решение;

y – решение задачи в точках t.

Приведем решение примера 6.2 с использованием функции solve_ivp ():

```
import numpy as np;import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
def func(t,y):
   dydt=np.sin(t)+y;
   return dydt
x2 = np.linspace(0,3,301) # интервал поиска решения
y1 = [2] # Значения функции  в точке x2[0]
y = solve_ivp(func,t_span =(0,3),y0=y1,t_eval=x2)
yt=-np.sin(x2)/2-np.cos(x2)/2+5/2*np.exp(x2)
plt.plot(x2,y.y.reshape(len(x2),1),':r*',x2,yt,'--g',lw=3,ms=4)
plt.legend(['Численное решение','Точное решение'])
p=np.max(abs((yt-y.y)))
print("Максимальная погрешность метода solve_ivp равна %.8f" % p)
plt.show()
```

Краткий комментарий к коду. Во избежание путаницы вместо переменной *x* использовалась переменная *t*. Результат работы программы – y.y. Для приведения y.y и x2 к одной размерности использовался метод reshape().

Результатом работы кода являются графическое решение задачи (рис. 6.3) и значение максимальной погрешности метода.

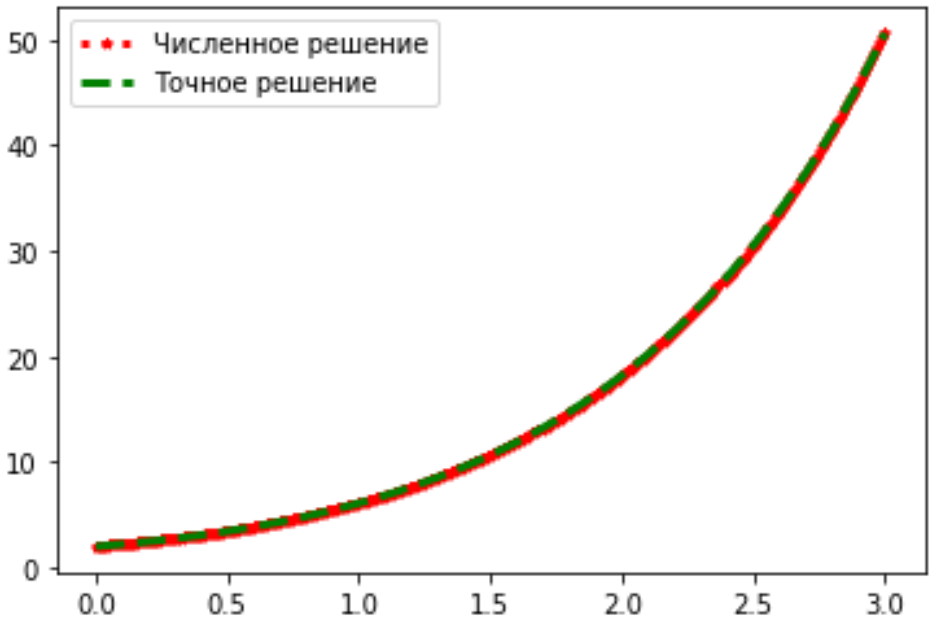

Рис. 6.3. Решение задачи методом solve_ivp

Максимальная погрешность метода solve_ivp равна 0.00306482.

Видим, что погрешность решения с автоматическим выбором метода решения больше, чем при решении методом odeint().

6.2. Решение задачи Коши для систем дифференциальных уравнений первого порядка

Для численного решения систем дифференциальных уравнений также можно использовать методы Эйлера и Рунге–Кутта. Покажем на примере, как можно численно решить задачу Коши для системы двух дифференциальных уравнений.

Пример 6.3. Решить задачу Коши для системы двух дифференциальных уравнений первого порядка методом Эйлера. Сделать три шага. Взять шаг равным 0.1. Сравнить полученное решение с точным решением системы дифференциальных уравнений:

$$\begin{cases} y' = z - 5\cos x \\ z' = 2y + z \\ y(0) = 4 \\ z(0) = 7 \end{cases}.$$

Точное решение системы:

$$\begin{cases} y(x) = 2e^{-x} + 3e^{2x} - 2\sin x - \cos x, \\ z(x) = -2e^{-x} + 6e^{2x} + \sin x + 3\cos x \end{cases}.$$

Решение. Запишем формулы метода Эйлера для поиска y_{i+1} и z_{i+1}:

$$\begin{cases} y_{i+1} = y_i + h(z_i - 5\cos x_i) \\ z_{i+1} = z_i + h(2y_i + z_i) \end{cases}.$$

Вычислим первую пару значений

$$\begin{cases} y_1 = y_0 + h(z_0 - 5\cos x_0) \\ z_1 = z_i + h(2y_0 + z_0) \end{cases}. \; x_0=0,\; y_0=4,\; z_0=7,\; h=0.1.$$

Тогда

$$\begin{cases} y_1 = 4 + 0.1(7 - 5\cos 0) = 4.2. \\ z_1 = 7 + 0.1(2 \cdot 4 + 7) = 8.5 \end{cases}.$$

$$\begin{cases} y_2 = y_1 + h(z_1 - 5\cos x_1) \\ z_2 = z_1 + h(2y_1 + z_1) \end{cases}, \; x_1 = x_0 + h.$$

$$\begin{cases} y_2 = 4.2 + 0.1(8.5 - 5\cos 0.1) = 4.5524979 \\ z_2 = 8.5 + 0.1(2 \cdot 4.2 + 8.5) = 10.260500 \end{cases}.$$

$$\begin{cases} y_3 = y_2 + h(z_2 - 5\cos x_2) \\ z_3 = z_2 + h(2y_2 + z_2) \end{cases}, \quad x_2 = x_1 + h.$$

$$\begin{cases} y_3 = 4.5524979 + 0.1(10.2605 - 5\cos 0.2) = 5.0885146 \\ z_3 = 10.2605 + 0.1(2 \cdot 4.5524979 + 10.2605) = 12.304252 \end{cases}.$$

Результаты запишем в виде таблицы

x	y (метод Эйлера)	y точное	z (метод Эйлера)	z точное
0	4	4	7	7
0.1	4,2	4.27921211	8,5	8.60358763
0.2	4.5524979	4.73553036	10.2605	10.45235574
0.3	5.0885146	5.40161594	12.304252	12.61260604

Видим, что по мере удаления от начальных точек погрешность вычислений возрастает.

Решим ту же задачу с помощью функций odeint() и solve_ivp().

Решение с использованием функции odeint():

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def F(s,x):
  dydx=s[1]-5*np.cos(x)
  dzdx=2*s[0]+s[1]
  return [dydx,dzdx]
s0=[4,7] # список - значения функций в нулевой точке
x = np.linspace(0, 2, 50);
s=odeint(F,s0,x) # ссылка на двумерный массив решения
plt.plot(x,s[:,0],'r',linewidth=3,label="y(x)")
plt.plot(x,s[:,1],'g--',linewidth=3,label="z(x)")
yt=2*np.exp(-x)+3*np.exp(2*x)-2*np.sin(x)-np.cos(x)
zt=-2*np.exp(-x)+6*np.exp(2*x)+np.sin(x)+3*np.cos(x)
plt.plot(x,yt,'k-',linewidth=2,label="yt")
plt.plot(x,zt,'m-*',linewidth=2,label="zt")
plt.xlabel("x"); plt.ylabel("y(x),     z(x)")
plt.legend(); plt.grid(); plt.show()
er1=np.max(abs(s[:,0]-yt));er2=np.max(abs(s[:,1]-zt)); er=max(er1,er2)
print("Максимальная погрешность метода odeint равна %.8f" % er)
```

В коде приняты обозначения: s[0] – это *y*, s[1] – это *z*.

Результатом работы кода являются графическое решение задачи (рис. 6.4) и значение максимальной погрешности метода.

Максимальная погрешность метода odeint равна 0.00001819.

И в этом примере численное решение практически совпало с точным. Значения *y* (s[:,0]) и *z* (s[:,1] при необходимости можно вывести на экран.

Решение с помощью функции solve_ivp():

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
def func(t,y):
   return [y[1]-5*np.cos(t),2*y[0]+y[1]]
   # t - это x
```

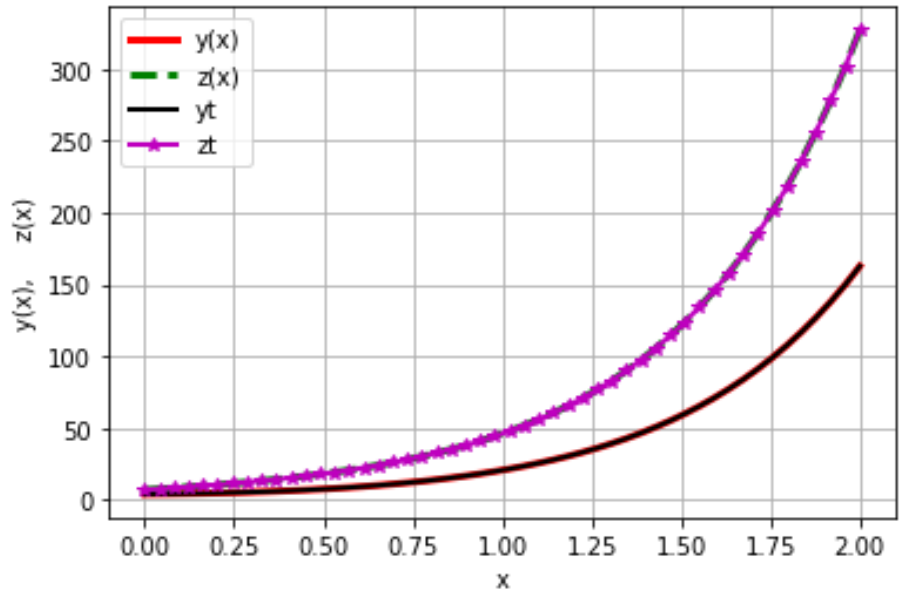

Рис. 6.4. Точное и приближенное решения задачи Коши для системы двух дифференциальных уравнений

```
    # y[0] - это y
    # y[1] - это z
s0=[4,7] # список - значения функций в нулевой точке
t = np.linspace(0,2,50); # точки для вычисления функций
sol = solve_ivp(func,t_span=(0,2),y0=s0,t_eval=t)
plt.plot(t,sol.y[0],'r',linewidth=3,label="y(x)")
plt.plot(t,sol.y[1],'g--',linewidth=3,label="z(x)")
yt=2*np.exp(-t)+3*np.exp(2*t)-2*np.sin(t)-np.cos(t)
zt=-2*np.exp(-t)+6*np.exp(2*t)+np.sin(t)+3*np.cos(t)
plt.plot(t,yt,'k-',linewidth=2,label="y точное")
plt.plot(t,zt,'m-*',linewidth=2,label="z точное")
plt.xlabel("x")
plt.ylabel("y(x),     z(x)")
plt.legend()
plt.grid()
plt.show()
er1=np.max(abs(sol.y[1]-zt))
er2=np.max(abs(sol.y[0]-yt))
er=max(er1,er2)
print("Максимальная погрешность метода solve_ivp равна %.8f" % er)
```

Результат:

Максимальная погрешность метода solve_ivp равна 0.09790382.
Графическое решение представлено на рис. 6.5.

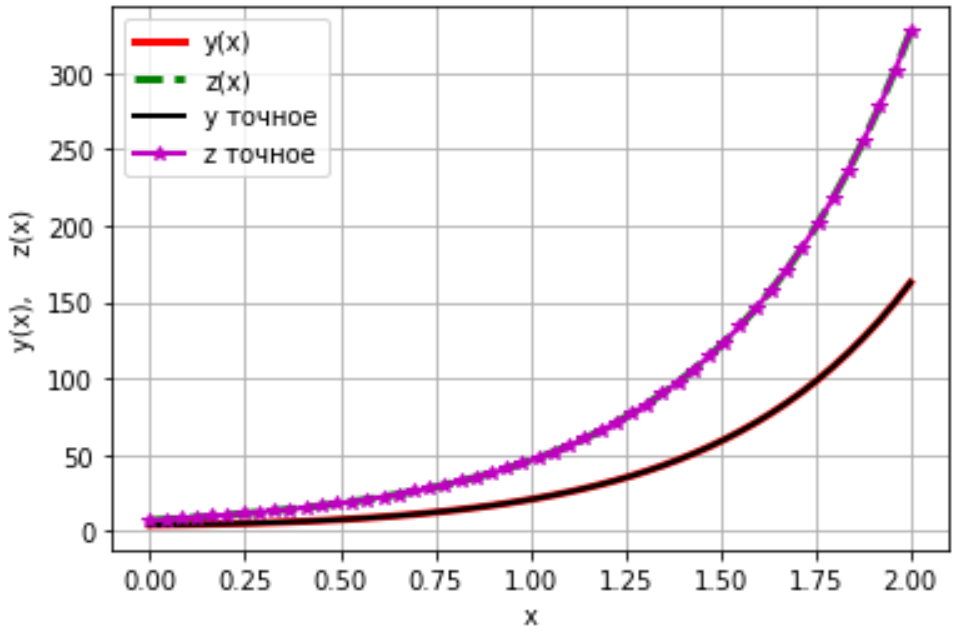

Рис. 6.5. Решение задачи Коши для системы двух дифференциальных уравнений методом solve_ivp()

И в этом случае погрешность получилась больше, чем при решении методом odeint().

6.3. Решение задачи Коши для дифференциальных уравнений высших порядков

Решение задачи Коши для дифференциальных уравнений высших порядков может быть сведено к решению задачи Коши для системы дифференциальных уравнений первого порядка. Покажем на примере, как это можно сделать.

Пример 6.4. Решить задачу Коши для дифференциального уравнения второго порядка:

$$\begin{cases} y'' - 2y' + 10y = e^{-x} \\ y(1) = 2 \\ y'(1) = 0.7 \end{cases}.$$

Решение найти с помощью функции odeint(). Сравнить результат с точным решением задачи. Построить графики точного и приближенного решений на отрезке [1, 3.5].

Решение. Сведем задачу решения дифференциального уравнения второго порядка к решению системы двух дифференциальных уравнений первого порядка: $\begin{cases} y'(x) = y_1(x) \\ y''(x) = y_1'(x) = 2y_1(x) - 10y(x) + e^{-x} \end{cases}$. При этом начальные условия перепишутся в виде $\begin{cases} y(1) = 2 \\ y_1(1) = 0.7 \end{cases}$. Такая замена нужна для численного решения задачи с помощью функции odeint().

Точное решение задачи в данном случае можно найти с помощью функции dsolve() библиотеки SymPy:

```
from sympy import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
x= symbols('x');f=Function('f')
eq=Eq(f(x).diff(x,2)-2*f(x).diff(x)+10*f(x),exp(-x));
yy=dsolve(eq,f(x));u=yy.rhs
yy0=u.subs({x:1})-2;# значение функции в точке 1
yy01=diff(u,x).subs({x:1})-.7 # значение производной в точке 1
C1,C2=symbols(" C1,C2")
sol_const=solve([yy0,yy01],[C1,C2])
w=yy.subs(sol_const);w1=w.rhs
def func(t,y):
   y0, y1 = y
# y0 - это решение дифуравнения, y1 - ее первая производная
# Возвращаются правые части системы дифуравнений первого порядка
   dydt = [y1, 2*y1-10*y0+exp(-t)]
   return dydt
x2 = np.linspace(1,3.5,30) # интервал поиска решения
y0 = [2.0, .7] # Значения функции и производной в точке x2[0]
y = odeint(func, y0, x2, tfirst=True)
# y[:,0] - численное решение дифуравнения в точках x2
# y[:,1] - вычисленное значение первой производной в точках x2
ff=lambdify(x,w1)
```

```
plt.plot(x2,y[:,0],':r*',x2,ff(x2),'-g',lw=3,ms=10)
plt.legend(['Численное решение', 'Точное решение'])
plt.show()
```

Графическое решение примера 6.4 представлено на рис. 6.6.

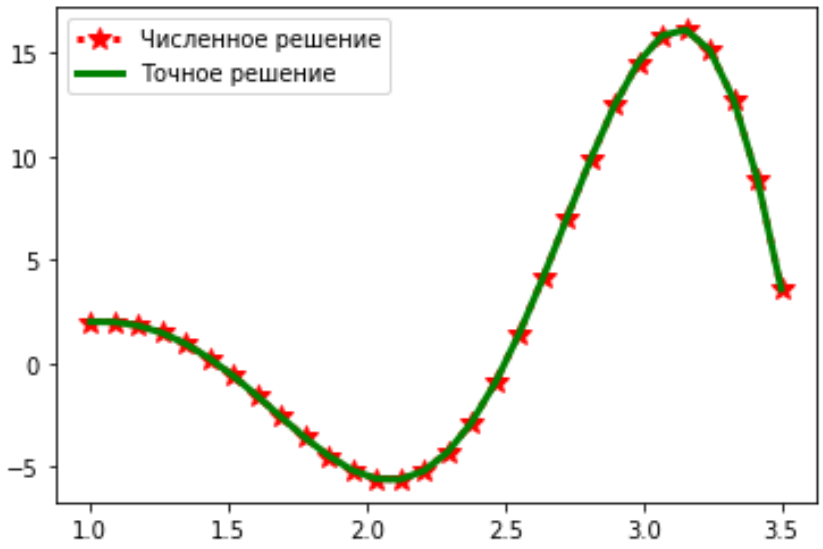

Рис. 6.6. Точное и численное решение задачи Коши для дифференциального уравнения второго порядка

Комментарий к коду. Подгружаем нужные библиотеки и функции. Поскольку точное решение мы находим с помощью библиотеки SymPy, объявляем нужные для решения задачи переменные (*x*, *C1* и *C2*) символьными, а переменную *f* – функцией. Создаем объект Eq, соответствующий дифференциальному уравнению eq1=Eq(f(x).diff(x,2)-2*f(x).diff(x)+10*f(x),exp(-x)). Если теперь вывести на экран значение eq1, получим математическую запись дифференциального уравнения:

$$10 f(x) - 2\frac{d}{dx} f(x) + \frac{d^2}{dx^2} f(x) = e^{-x}.$$

Решаем дифуравнение с помощью функции dsolve(). Если вывести на экран решение *yy*, получаем $f(x) = \left(c_1 \sin(3x) + c_2 \cos(3x)\right)e^{x} + \frac{e^{-x}}{13}$. *yy0* – это значение функции в точке *x0*, а *yy01* – значение первой производной в этой же точке. Коэффициенты c_1 и c_2 находим из решения системы двух линейных уравнений с двумя неизвестными. В коде это словарь sol_const. Значения переменных C1 и C2: {C1: 0.253309450733069, C2: -0.696572394820176}. Решение задачи Коши – переменная w. Вывод ее на экран дает

$$f(x) = (0.253309450733069 \sin(3x) - 0.696572394820176 \cos(3x))\, e^x + \frac{e^{-x}}{13}$$

w1 – это правая часть этого уравнения. Далее находим численное решение дифференциального уравнения. Для этого создаем функцию func(t,y) с правыми частями системы дифференциальных уравнений первого порядка. В коде *y0* это – $y(x)$, в коде *y1* это – $y_1(x)$. Задаем интервал поиска решения, начальные условия и решаем задачу с помощью функции odeint(), передав в нее необходимые параметры. Решение задачи – вектор y[:,0]. Для вычисления точного решения в заданных точках создаем лямбда-функцию *ff*. Наносим на график точное и приближенное решения. Из рис. 6.6 видно, что решения практически совпадают.

Данную задачу можно решить и с использованием функции solve_ivp ().

Задания для самостоятельной работы

Задание 1. Решить задачу Коши для ОДУ первого порядка:

	$xdy = (y + \frac{\nu}{x})dx$	$xy' = y + x^2 \cos x$
а)	$x_0 = 3 + \nu + 0.1\mu$;	б) $x_0 = 0.5(\mu + \theta + 1)$.
	$y_0 = 2 + \theta + 0.1\gamma$	$y_0 = (\nu + \gamma)/2$

Сделать три шага (h = 0.2) методами Эйлера и Рунге–Кутта. Результаты свести в таблицу. Дать графическое решение задачи на интервале [x_0, x_0+2] с тем же шагом с помощью функции solve_ivp().

Задание 2. Решить задачу Коши для системы двух дифференциальных уравнений первого порядка на интервале [0, 10], решение получить с шагом 0.5:

$$\begin{cases} y' = \cos(N_{cm} \cdot y + z) \\ z' = \sin(y + N_{cm} \cdot xz) \\ y(0) = 1 \\ z(0) = 2 \end{cases}.$$

Здесь N_{cm} – номер студента по списку. Для решения использовать функцию odeint().

Задание 3. Решить задачу Коши для дифференциального уравнения второго порядка:

$$\begin{cases} y'' - 6y' - \nu y = (\mu + 1)x \\ y(0) = 1 \\ y'(0) = 0 \end{cases}.$$

Решение найти с помощью функции odeint(). В библиотеке SymPy получить точное решение задачи. Сравнить точное и приближенное решения. Построить графики точного и приближенного решений на отрезке [0, 2].

ЛИТЕРАТУРА

1. Документация по NumPy и SciPy: сайт. – URL: https://docs.scipy.org/doc/.

2. Решаем систему линейных алгебраических уравнений с Python-пакетом scipy.linalg: сайт. – URL: https://habr.com/ru/company/macloud/blog/564154/.

3. Титов, А. Н. Решение математических задач в интегрированной среде Scilab: учеб.-метод. пособие / А. Н. Титов, Р. Ф. Тазиева. – Казань: Изд-во КНИТУ, 2022. – 164 с.

4. Титов, А. Н. Визуализация данных в Python. Работа с библиотекой Matplotlib: учеб.-метод. пособие / А. Н. Титов, Р. Ф. Тазиева. – Казань: Изд-во КНИТУ, 2022. – 92 с.

5. Арушанян, И. О. Численные алгоритмы решения нелинейных уравнений: учеб.-метод. пособие / И. О. Арушанян. – Москва: Изд-во МГУ, 2018. – 39 с.

6. Моделирование в энергетике: сайт. – URL: http://simenergy.ru/math-analysis/solution-methods/44-method-ridders.

7. Самарский, А. А. Численные методы / А. А. Самарский, А. В. Гулин. – Москва: Наука, 1989. – 432 с.

8. Мудров, В. И. Метод наименьших модулей / В. И. Мудров, В. Л. Кушко. – Москва: Знание, 1971. – 64 с.

9. Калиткин, Н. Н. Интерполяция *B*-сплайнами / Н. Н. Калиткин, Н. М. Шляхов // Математическое моделирование. – 2022. – Т. 14, № 4. – С. 109–120.

10. Сеть RBF: сайт. – URL: https:// russianblogs.com /article/6437535116/.

11. Радиальные нейронные сети: сайт. – URL: https://www.sgu.ru/sites/default/files/textdocsfiles/2013/12/10/lekciya_11.pdf.

12. Даутов, Р. З. Численные методы. Приближение функций: учеб. пособие / Р. З. Даутов, М. Р. Тимербаев. – Казань: Изд-во Казан. ун-та, 2021. – 123 с.

13. Решение задач по численным методам: сайт. – URL: https://elar.urfu.ru/bitstream/10995/1019/1/umk_2005_012.pdf.

УЧЕБНОЕ ИЗДАНИЕ

Андрей Николаевич Титов
Рамиля Фаридовна Тазиева

РЕШЕНИЕ ЗАДАЧ ЛИНЕЙНОЙ АЛГЕБРЫ И ПРИКЛАДНОЙ МАТЕМАТИКИ В PYTHON

РАБОТА С БИБЛИОТЕКОЙ SCIPY

Редактор Л. А. Муравьева
Компьютерная верстка и макет – А. Н. Егоров

Подписано в печать 01.06.2023 Формат 60×84 1/16
Бумага офсетная Печать цифровая 7,20 усл. печ. л.
7,75 уч.-изд. л. Тираж 400 экз. Заказ 35/23

Издательство Казанского национального исследовательского технологического университета

Отпечатано в офсетной лаборатории Казанского национального исследовательского технологического университета

420015, Казань, К. Маркса, 68

124