Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Ярославский государственный технический университет»

М.Е. Соловьев

СТАТИСТИЧЕСКИЙ АНАЛИЗ И ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ В PYTHON

Рекомендовано научно-методическим советом университета
в качестве учебного пособия

Ярославль
2025

УДК 519
ББК 22.172
С60

Соловьев, М.Е.

С60 **Статистический анализ и имитационное моделирование в Python**: учебное пособие / М.Е. Соловьев – Ярославль: Изд. дом ЯГТУ, 2025. – 388 с.

ISBN 978-5-9914-0437-2

Рассмотрены методы статистического анализа и информационного моделирования с использованием универсального языка программирования Python.

Предназначено для студентов, обучающихся по направлению «Информационные системы и технологии» и аспирантов по научной специальности «Математическое моделирование, численные методы и комплексы программ»

Ил. 18. Табл. 1. Библиогр. 31.

УДК 519
ББК 22.172

Рецензенты: Е.П. Кубышкин, д-р физ.-мат. наук, профессор ЯрГУ им. П.Г. Демидова.

ISBN 978-5-9914-0437-2

Содержание

Введение

Статистический анализ представляет собой методику, используемую для сбора, обработки и интерпретации данных с целью выявления закономерностей и тенденций. Задачи статистического анализа встречаются в самых различных областях: медицине и биологии, химии и химической технологии, машиностроении, строительстве. В информационных технологиях статистический анализ широко применяется при обработке больших объемов данных, в нейронных сетях и машинном обучении. Математической основой статистического анализа является теория вероятностей и математическая статистика [1, 2, 3]. В процессе статистического анализа производится исследование распределений выходных параметров исследуемой системы, оценка тесноты статистической связи между ее входными и выходными переменными, решается задача поиска оптимальных значений входных переменных, обеспечивающих минимальное или максимальное (в зависимости от постановки задачи) значение функции отклика.

Широкому распространению методов статистического анализа способствовало развитие вычислительной техники и соответствующего программного обеспечения. На сегодняшний день создано большое количество специализированных программных средств для автоматизации статистического анализа. Среди открытых пакетов для статистического анализа наиболее популярным является программный пакет R [4], являющийся интерпретируемым языком программирования, предназначенным для статистической обработки данных и работы с графикой. Мощным средством для решения задач статистического анализа являются также специализированные библиотеки языка Python. Python широко используется для статистического анализа данных благодаря своей простоте, большому количеству библиотек и возможностям синтаксиса для построения специализированных приложений. Библиотеки Python позволяют легко подключаться к различным источникам данных, очищать и преобразовывать данные перед анализом.

К основным библиотекам Python для решения задач статистического анализа относятся следующие.

1. Pandas [5] - является основной библиотекой для обработки и анализа данных. Она предоставляет структуры данных, такие как DataFrame, которые упрощают манипуляцию с данными, включая их очистку и подготовку для анализа

2. NumPy [6] - используется для работы с многомерными массивами и предоставляет функции для выполнения сложных математических и статистических расчетов. Это основа для многих других библиотек, включая Pandas.

3. SciPy [7] - расширяет возможности NumPy и включает в себя дополнительные функции для выполнения научных и инженерных расчетов, таких как интеграция, оптимизация и статистические тесты. Модуль scipy.stats, предоставляет множество статистических функций для анализа данных, таких как расчет t-критерия, корреляции Пирсона, дисперсионного анализа, регрессионного анализа и др.

4. StatsModels [8]. Данная библиотека предназначена для выполнения статистического моделирования и анализа. Она предоставляет классы и функции для оценки различных статистических моделей и проведения тестов гипотез.

5. Matplotlib [9] и Seaborn [10]. Эти библиотеки используются для визуализации данных. Matplotlib позволяет создавать различные графики, а Seaborn строит более сложные визуализации с учетом статистических аспектов данных.

Методы статистического анализа позволяют исследовать исторические данные и делать выводы о вероятностных характеристиках процессов. Однако, их возможности ограничены, когда речь идет о сложных системах с множеством взаимодействующих переменных, где простые статистические методы могут не дать полного представления о динамике системы. Расширение возможностей статистического анализа достигается за счет использования имитационного моделирования систем. Имитационное моделирование, в свою очередь, представляет собой более сложный подход, который позволяет создавать модели реальных систем и проводить эксперименты с этими моделями. Этот метод позволяет учитывать взаимодействия между различными компонентами системы и их влияние на результаты. В процессе имитационного моделирования исследователь может варьировать параметры и наблюдать за изменениями в системе, что дает возможность выявить оптимальные условия для функционирования системы. При этом собственно статистический анализ может использоваться на различных этапах имитационного моделирования. В частности, статистический анализ используется для подготовки исходных данных, которые будут служить основой для имитационной модели. Это может включать анализ исторических данных, выявление закономерностей и определение ключевых переменных, влияющих на систему. Также методы статистического анализа используются при верификации и валидации модели. Результаты

экспериментов на модели анализируются с использованием методов статистического анализа и на их основе принимаются решения о том, как улучшить функционирование системы.

Существует множество программных пакетов для имитационного моделирования, каждый из которых предлагает различные инструменты и возможности для анализа и оптимизации сложных систем. К наиболее популярным из них относятся такие пакеты как AnyLogic [11], NetLogo [12], PySCeS [13], Simul8 [14]. Старейшим инструментальным средством имитационного моделирования является GPSS (General Purpose Simulation System) [15] - язык программирования, специально разработанный для имитационного моделирования. Вместе с тем, имеются и специализированные библиотеки Python, предназначенные для имитационного моделирования. К наиболее популярным можно отнести следующие библиотеки.

1. SimPy [16] - это библиотека для моделирования процессов на основе событий. Она позволяет создавать модели, которые описывают системы, функционирующие во времени, и подходит для моделирования процессов в таких областях, как производство, транспорт и связь.

2. Salabim [17] - также библиотека дискретно-событийного моделирования, имеющая широкий спектр областей применения. В отличие от SimPy данная библиотека не основана генераторах Python для управления процессами.

3. PySD [18] - библиотека для создания моделей системной динамики на Python. Она переводит модели из формата Vensim в Python, что позволяет использовать возможности Python для анализа и визуализации. PySD поддерживает интеграцию с инструментами больших данных и машинного обучения, что делает её полезной для современных приложений.

4. BPTK_Py [19] - также библиотека для создания имитационных моделей системной динамики. Она позволяет строить модели с использованием stocks, flows и converters, а также интегрируется с Jupyter для интерактивного моделирования. Библиотека предоставляет функции для визуализации и анализа результатов симуляций.

Этот список может быть продолжен. В работе [20] приводится сравнение возможностей открытых библиотек Python с одним из популярных коммерческих пакетов для имитационного моделирования ARENA [21].

Таким образом, статистический анализ и имитационное моделирование - это две взаимосвязанные области. В настоящем пособии приведены в необходимом объеме теоретические

сведения в каждой из этих областей и примеры решения задач и моделирования с использованием библиотек Python.

Глава 1. Введение в язык программирования Python

Python сегодня входит в пятерку наиболее популярных языков программирования общего назначения, обходя C/C++ и многие другие известные алгоритмические языки. Этому способствуют широкие возможности языка, его доступность и простота в изучении. Если характеризовать кратко, то это - объектно-ориентированный, интерпретируемый, переносимый язык сверхвысокого уровня. Вот некоторые его важнейшие положительные качества:

1) динамическая типизация и интроспекция устраняет необходимость в описании переменных или аргументов;

2) высокоуровневые типы данных позволяют выражать сложные операции в одной инструкции;

3) простой и изящный синтаксис и автоматическое управление памятью облегчает написание и читаемость программ;

4) высокая степень переносимости кода позволяет практически не заботиться о конечной платформе;

5) поддержание нескольких парадигм программирования: императивное (процедурный, структурный, модульный подходы), объектно-ориентированное и функциональное программирование обеспечивает гибкость языка для решения задач самых разных классов;

6) доступность: возможности использования как в проектах Open Source, так и коммерческих программах.

7) огромное количество написанных к настоящему времени библиотек и модулей, имеющихся в открытом доступе облегчает создание новых программ.

Первоначально Python был создан Гвидо ван Россумом (Guido van Rossum) в 1991 году как язык управления сценариями. В дальнейшем к его разработке подключилось большое сообщество программистов, так что в нем реализованы самые современные идеи в технологии программирования. В настоящее время Python быстро развивается, постоянно растет количество поддерживающих его платформ.

Как интерпретируемый язык, Python имеет определенные ограничения, связанные со скоростью выполнения программ. Однако при решении задач математического моделирования эти

ограничения в значительной степени снимаются путем использования прикладных пакетов, таких как Numpy и Scipy, в которых реализованы специальные структуры данных и имеется большое количество откомпилированных функций для решения математических и научных задач. Кроме того, создан язык программирования Cython, упрощающий написание модулей C/C++ кода для Python. Его возможности могут быть использованы в тех случаях, когда скорость выполнения программы является критическим фактором.

Начать работу в Python можно без установки его на компьютер. Существует много онлайн-интерпретаторов, позволяющих программировать непосредственно в браузере. Удобным онлайн-интерпретатором является Google Colab, или Colaboratory (https://colab.research.google.com/), представляющий собой бесплатную облачную платформу от Google, которая позволяет пользователям писать и выполнять код на Python в среде, аналогичной Jupyter Notebook (https://jupyter.org/). Вместе с тем при систематической работе целесообразно иметь интерпретатор и основные библиотеки установленными на компьютер. Интерпретатор Python включен практически во все современные дистрибутивы Linux, так что для запуска его достаточно в командной строке терминала набрать:

```
python3
```

После этого в окне отобразится информация о текущей версии языка и появится строка приглашения ">>>", в которой можно набирать команды интерпретатору. Для получения справочной информации на английском языке можно набрать в строке приглашения `help()` и программа перейдет в режим помощи, при этом вид строки приглашения изменится на `help>`. Теперь можно получить документацию по всем разделам. Например, можно вызвать список ключевых слов языка, набрав в строке приглашения `keywords`. Для выхода из режима помощи наберите `quit`. Для выхода из интерпретатора необходимо ввести команду `exit()`.

Для установки Python на Windows перейдите на официальный сайт Python и скачайте последнюю версию установочного файла для Windows (формат *.exe). Вместе с интерпретатором устанавливается среда программирования IDLE, в которую входит командное окно интерпретатора и редактор текстовых файлов для написания и редактирования программ.

Учитывая, что по языку Python имеется достаточно много литературы мы не будем подробно останавливаться на синтаксисе, а непосредственно перейдем к примерам.

1.1. Работа в интерактивном режиме. Типы структурированных данных

Рассмотрим работу программы в режиме калькулятора. Наберите в строке приглашения арифметическое выражение и нажмите "Enter":

```
>>> 34+75*3
259
```

Если по условиям синтаксиса языка ввод должен занимать более одной строки, то вид приглашения для последующих строк изменяется на "...". Например, введите строку комментария, начинающуюся с символа "#":

```
>>> # Это комментарий
...# А это продолжение
...
```

Чтобы закончить продолжающийся ввод нажмите "Enter".

Python является строго типизированным языком. Динамическая типизация не отменяет необходимости следить за согласованием типов в выражениях. Например, в версиях Python до 3.0 ввод в строке приглашения:

```
>>> 7/3
2
```

дает результат целого типа ("int"), путем преобразования частного к типу операндов, заданных в данном случае целыми (округление производится в меньшую сторону: 7/-3 даст -3). Это может служить источником ошибок для новичков в программировании. Для получения вещественного результата необходимо указывать операнды как вещественные числа (тип "float", что соответствует типу "double" C/C++):

```
>>> 7.0/3.0
 2.3333333333333335
```

или, по крайней мере, один из операндов должен иметь тип "float":

```
 >>> 7.0/3
 2.3333333333333335
```

Следует заметить, что для избежания проблем, связанных с преобразованием типов при делении в версиях Python начиная с 3.0 оператор деления «/» сделан всегда отвечающим истинному делению (с сохранением дробной части числа) вне зависимости от типа операндов. Для целочисленного деления имеется оператор «//»:

```
>>> 1//2
0
```

Для преобразования типов имеется достаточно большое количество функций. К примеру, часто возникает необходимость в преобразовании чисел в строки и наоборот. Определите строковую переменную:

```
>>> Mes="Result of division 7/3 is "
```

Для вывода ее значения достаточно набрать в строке приглашения идентификатор и нажать "Enter":

```
>>> Mes
'Result of division 7/3 is '
```

Определите переменную типа "int"

```
>>> a=7/3
```

В данном случае мы имеем пример динамической типизации. Тип переменной "a" определяется в процессе вычисления. Попробуйте выполнить конкатенацию:

```
>>>Mes+a
```

Интерпретатор выведет сообщение об ошибке:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str
```

Мы попытались выполнить операцию над переменными не согласующихся типов. Правильный результат получится, если переменную "a" предварительно преобразовать к строковому типу:

```
>>> Mes+str(a)
```

```
'Result of division 7/3 is 2.3333333333333335'
```

Для математических вычислений в Python имеются необходимые функции, однако встроенными являются лишь небольшое число функций:

```
>>> abs(-5) #   Модуль числа
5
>>> divmod(14.0, 4) # Частное и остаток от деления. Результат: (частное, остаток)
(3.0, 2.0)
>>> pow(3.0, 4) # Возведение  в степень
81.0
>>> 3.0**4  # Альтернативный вариант возведения  в степень
81.0
```

Но если Вы попытаетесь, например, вычислить тригонометрическую функцию, то получите сообщение об ошибке:

```
>>> sin(3.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

Большинство необходимых на практике функций определено в пакетах. Загрузка пакета в память осуществляется с помощью инструкции import. Загрузим пакет math для математических функций (для работы с комплексными числами используется пакет cmath) :

```
>>> import math
```

С полным описанием определенных в нем функций можно ознакомиться по справке (в режиме справки набрать в строке приглашения math). Доступ к конкретной функции осуществляется с помощью механизма "квалифицированных имен", принятого в C++:

```
>>> math.sin(3.0)
0.14112000805986721
```

В этом пакете определены также константы "e" (основание натурального логарифма) и "pi":

```
>>> math.e
2.7182818284590451
```

```
>>> math.pi
3.1415926535897931
>>> math.sin(math.pi)
1.2246467991473532e-16
```

Как видно, результат получается с точностью, принятой для чисел типа float, который отвечает типу double в C/C++.

Несколько замечаний об именах переменных в Python. Имена переменных должны начинаться с символа подчеркивания или с алфавитного символа, за которым может следовать произвольное число алфавитно-цифровых символов и символов подчеркивания. Регистр символов в именах имеет значение. Запрещено использовать в качестве имен переменных зарезервированные слова. Список зарезервированных слов можно получить в режиме интерактивной справки, набрав в приглашении help> ключевое слово keywords. Относительно символов подчеркивания в начале и в конце имени переменных имеются дополнительные соглашения: имена переменных с двумя символами подчеркивания в начале и в конце (например, __name__) обычно имеют особый смысл для интерпретатора, поэтому не следует их использовать; имена переменных, начинающихся с двух символов подчеркивания и не оканчивающихся на них (например, __X) являются локальными при определении в классе и автоматически подменяются другими для объемлющего класса; имена переменных, начинающихся с одного символа подчеркивания не импортируются при использовании инструкции «from module import *».

Перейдем к рассмотрению новых структурированных типов данных, реализованных в Python. Прежде всего - это списки. Список выглядит похожим на массив в традиционных алгоритмических языках. Определим две переменные типа списков.

```
>>> a=[1,2,3]
>>> b=[4,5,6,7]
```

Внешне это немного напоминает определение и инициализацию массива. Индексация списков, также как и массивов в C/C++ начинается с 0. Аналогично с массивом выглядит и способ доступа к элементу списка по индексу элемента.

```
>>> a[0]
1
>>> b[1]
```

```
5
```

Но на этом, пожалуй, аналогия заканчивается:

```
>>> a+b
[1, 2, 3, 4, 5, 6, 7]
```

Перегруженный оператор сложения представляет собой не сумму массивов, которая в данном случае была бы не возможной из-за различия в их размерности, а конкатенацию, аналогично тому как он работает для строк. Вместе с тем для числовых элементов списков знак суммы сохраняет свое традиционное значение:

```
>>> a[0]+b[0]
5
```

Теперь уже можно догадаться, что оператор умножения по отношению к спискам также работает не традиционно:

```
>>> a*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Эффект динамической типизации для списков особенно выразителен: в одном списке можно объединять объекты разных типов:

```
>>> s=['one','two','three']
>>> s+b
['one', 'two', 'three', 4, 5, 6, 7]
```

И, что замечательно, каждый из них поддерживает при этом свои методы:

```
>>> t=s+b
>>> t[2]=t[2]+' and four'
>>> t[3]=t[3]+3
>>> t
['one', 'two', 'three and four', 7, 5, 6, 7]
```

Но, естественно, путать элементы разных типов нельзя:

```
>>> s[0]+b[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

Для доступа одновременно к нескольким элементам списка используется механизм срезов:

```
>>> t[0:2]
['one', 'two']
```

Первая цифра в этом обозначении среза указывает начальный индекс, а вторая индекс, на единицу превышающий индекс последнего вырезаемого элемента. Первый индекс "0" можно опускать:

```
>>> t[:3]
['one', 'two', 'three and four']
```

Отрицательное значение второго индекса в спецификации используется для отсчета элементов с конца последовательности:

```
>>> t[:-4]
['one', 'two', 'three and four']
```

Срезы являются эффективным средством для работы с последовательностями, в частности со списками:

```
>>> t[:2]=a[:2] # Заменить несколько элементов
>>> t
[1, 2, 'three and four', 7, 5, 6, 7]
>>> t[1:3]=[] # Удалить
>>> t
[1, 7, 5, 6, 7]
>>> t[2:2]=3*a[:2] # Вставить
>>> t
[1, 7, 1, 2, 1, 2, 1, 2, 5, 6, 7]
>>> b[:0]=b # Вставить копию самого себя в начало
>>> b
[4, 5, 6, 7, 4, 5, 6, 7]
```

Некоторые функции и методы для работы со списками:

```
>>> len(b) # Определение количества элементов
```

```
8
>>> s.append('four') # Добавление элемента в конец списка
>>> s
['one', 'two', 'three', 'four']
```

Стоит заметить, что метод "append" при передаче в качестве параметра списка не представляет собой конкатенацию двух списков. Он добавляет объект целиком в качестве элемента списка. Так:

```
>>> b.append([8,9]) # Добавление объекта в конец списка
>>> b
[4, 5, 6, 7, 4, 5, 6, 7, [8, 9]]
>>> len(b)
9
```

То есть мы таким образом получили вложенный список.

Доступ к элементам вложенного списка осуществляется по индексам:

```
>>> b[9]
[8, 9]
>>> b[9][1]
9
```

Особенностью метода "append" является также то, что он не возвращает нового объекта списка, а изменяет существующий. Попытка присвоить переменной результат действия этого метода приведет, как правило, к уничтожению списка:

```
>>>b=b.append([10,11])
>>>print b
None
```

Приведем еще некоторые полезные методы работы со списками.

Определение индекса с заданным значением

```
>>> s.index('three')
2
>>> t.sort() # Сортировка по возрастанию
```

```
>>> t
[1, 1, 1, 1, 2, 2, 2, 5, 6, 7, 7] .
```

Приведенные функции составляют далеко не полный перечень, но даже они показывают, насколько широкие возможности предоставляет этот тип данных.

Похожие функции Python поддерживает и при работе со строковыми переменными. В частности для строк также работает механизм срезов, можно взять элемент строки по индексу. Однако, в отличие от списков, строки представляют собой неизменяемые последовательности:

```
>>>s='stroka'
>>>ls=list(s)
>>> s[0]
's'
>>> ls[0]
's'
>>> del(ls[0])
>>> ls
['t', 'r', 'o', 'k', 'a']
>>> del(s[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion
```

Аналогичным списку встроенным типом последовательностей является кортеж (tuple). В отличие от списков кортеж, также как и строка, - неизменяемая последовательность. Он выглядит похожим на список - представляет собой последовательность элементов, разделенных запятой, но заключенных не в квадратные, а в круглые скобки. В тех случаях, когда не возникает неоднозначность в интерпретации, скобки при записи кортежа вообще могут быть опущены:

```
>>> k=100, 101, 'hundred'
>>> k
(100, 101, 'hundred')
```

Кортежи применяются в тех случаях, когда по смыслу программы требуется применение списка, но необходимо, чтобы его элементы были не изменяемыми. Иногда требуется кортеж, не

включающий ни одного элемента, или кортеж, состоящий только из одного элемента. При их определении необходимы некоторые ухищрения:

```
>>> o=()
>>> type(o)
<type 'tuple'>
>>> e=1,
>>> type(e)
<type 'tuple'>
>>> o
()
>>> e
(1,)
```

Еще один встроенный тип структурированных данных, который также находит широкое применение, - словарь (dictionary). В отличие от последовательностей, где элементы индексируются натуральным числами, в словарях элементы индексируются ключом, в качестве которого используются значения любого типа, не допускающего изменений. Такими элементами, в частности, являются числа, строки и кортежи. Причем кортежи могут быть только такие, которые состоят из неизменяемых объектов. Списки не могут выполнять роль ключей в словаре, поскольку их элементы можно изменять. Словарь может быть определен как неупорядоченное множество пар "ключ:значение", с требованием уникальности всех ключей в пределах одного словаря. Множество заключается в фигурные скобки, в котором пары разделяются запятыми. Пример определения словаря:

```
>>> d1={'Ivanov':'student','Petrov':'student','Sidorov':'professor','Smirnov':'docent'}
```

Доступ к элементам словаря производится не по индексу, как в последовательности, а по ключу:

```
 >>> d1['Ivanov']
'student'
>>> d1['Smirnov']
'docent'
```

Список ключей и соответствующих им значений можно получить с помощью методов keys() и values() соответственно:

```
>>> d1.keys()
['Ivanov', 'Petrov', 'Sidorov', 'Smirnov']
>>> d1.values()
['student', 'student', 'professor', 'docent']
```

Проведение проверки на вхождение элемента в словарь обеспечивает функция in:

```
>>> 'professor' in salary
True
```

Итак, Python предоставляет программисту целый набор структурированных типов данных, для которых определены полезные функции и методы. Однако при работе с такими данными нужно быть очень аккуратными. Рассмотрим, на первый взгляд, вполне безобидный пример. Определим переменную "a" как список из трех чисел

```
>>> a=[1,2,3]
```

Создадим на ее основе еще одну переменную

```
>>> b=a
>>> b
[1, 2, 3]
```

Все, вроде бы, хорошо. Будем изменять элементы b. Например, так

```
>>> b[0]=b[0]+10
>>> b
[11, 2, 3]
```

Однако, если теперь вызвать переменную "a", обнаружим следующее:

```
>>> a
[11, 2, 3]
```

Мы изменяли переменную "b", но элементы переменной "a" тоже при этом изменились без всякого нашего участия!

Причина состоит в том, что имена переменных в Python представляют собой лишь ссылки на объекты в памяти компьютера. Операцией "b=a" мы не создаем новый объект, а создаем лишь новую ссылку на один и тот же объект в памяти. При этом мы можем изменять объект совершенно равноправно либо ссылаясь на него с помощью "a", либо при помощи ссылки "b". Как же быть, когда нам реально нужен новый объект, а не новая ссылка на тот же самый? Для этой цели имеется специальная функция копирования объектов - "copy". Она расположена в специальном модуле с таким же именем "copy", поэтому вызов ее немного выглядит тавтологией: x = copy.copy(y). Модуль должен быть предварительно импортирован. Повторим теперь наши эксперименты:

```
>>> import copy
>>> a=[1,2,3]
>>> b=copy.copy(a)
>>> b
[1, 2, 3]
>>> b[0]=b[0]+10
>>> b
[11, 2, 3]
>>> a
[1, 2, 3]
```

Теперь как-то лучше. Но и это еще не все. Метод copy() создает так называемую "неглубокую" или "поверхностную" копию объекта ("shallow copy"). К сожалению, для некоторых сложных объектов даже этого оказывается мало. Для таких критических случаев создана специальная функция deepcopy(), которую рекомендуется использовать, когда элементами структурированного объекта также являются структурированные объекты. Например, если элементом списка является другой список, то функция copy() оказывается бессильна:

```
>>> a
[1, 2, 3]
>>> c=[4,5,6]
>>> a.append(c)
>>> a
[1, 2, 3, [4, 5, 6]]
```

```
>>> a[3][1]
5
>>> b=copy.copy(a)
>>> b
[1, 2, 3, [4, 5, 6]]
>>> b[3][1]=b[3][1]+10
>>> b
[1, 2, 3, [4, 15, 6]]
>>> a
[1, 2, 3, [4, 15, 6]]
```

И только deepcopy() спасает положение!

```
>>> b=copy.deepcopy(a)
>>> b
[1, 2, 3, [4, 15, 6]]
>>> b[3][1]=b[3][1]-10
>>> b
[1, 2, 3, [4, 5, 6]]
>>> a
[1, 2, 3, [4, 15, 6]]
```

Заметим, что необычные свойства, проявляющиеся в рассмотренных выше примерах, обусловлены тем, что переменные в Python представляют собой лишь ссылки на объекты в памяти. Сами объекты при этом могут быть изменяемыми, например, списки и словари и неизменяемыми, например, числа, строки и кортежи. Если две или несколько переменных ссылаются на один и тот же изменяемый объект, то при изменении его компонентов изменения отражаются на всех переменных. В случае неизменяемых объектов этого не происходит:

```
>>> s1='abc'
>>> s2=s1
>>> s1='abc'+'d'
>>> s1
'abcd'
```

```
>>> s2
'abc'
```

В данном случае инструкцией «s1='abc'+'d'» создается новый объект в памяти, а не изменение исходного. В то же время переменная s2 продолжает ссылаться на старый объект 'abc'.

Следующий пример демонстрирует суть переменных в Python как ссылок:

```
>>> L1=(1,('a','b','c'))
>>> L2=(1,('a','b','c'))
>>> L1==L2,L1 is L2
(True, False)
```

В данном случае значения переменных L1 и L2 равны, но они представляют собой ссылки на разные объекты в памяти. Здесь, вместе с тем, имеется некоторая тонкость, состоящая в том, что интерпретатор может сохранять в кэше небольшие числа, так что может оказаться, что разные переменные будут представлять ссылки на один и тот же объект, например:

```
>>> A_litle=40
>>> a_litle=40
>>> A_litle==a_litle, A_litle is a_litle
(True, True)
```

Тогда как:

```
>>> A_big=123E15
>>> A_big
1.23e+17
>>> a_big=123E15
>>> a_big
1.23e+17
>>> A_big==a_big, A_big is a_big
(True, False)
```

При работе с изменяемыми объектами, таким как списки, необходимо внимательно подходить к выбору способа изменения. Так, в частности, операция конкатенации списков не изменяет объект, а создает новый:

```
>>> A=[1,2,3]
```

```
>>> B=A
>>> A=A+[4,5,6]
>>> A,B
([1, 2, 3, 4, 5, 6], [1, 2, 3])
```

В то же время операция дополняющего присваивания «+=» с точки зрения синтаксиса языка C должна дать эквивалентный результат, однако на самом деле она не создает новый объект, а изменяет исходный:

```
>>> A=[1,2,3]
>>> B=A
>>> A+=[4,5,6]
>>> A,B
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6])
```

По производительности дополняющее присваивание работает быстрее, но надо иметь в виду разницу в механизмах этих двух способов изменения переменных.

1.2. Определение функций. Управляющие конструкции

Работа в интерактивном режиме полезна в основном для изучения возможностей языка, а также отладки небольших участков кода. Практический интерес представляет создание программ. Программа в Python представляет собой обычный текстовый файл с инструкциями, которому принято давать расширение ".py". Простейший вариант запуска программы представляет собой команду в терминале:

```
python <имя файла программы>
```

Другой вариант - присвоить файлу атрибут исполняемого и запускать его обычным образом, как другие исполняемый файлы, из окна файлового менеджера.

Перейдем к рассмотрению конкретных программ. Поскольку мы уже немного познакомились с работой интерпретатора, начнем, с чего-то более содержательного по сравнению с традиционным "Hello world!". Ниже представлена программа, выводящая на печать первые 11 чисел ряда Фибоначчи (ряд, в котором каждое последующее число равно сумме двух предыдущих).

```
def fib(n):
    if n>1:
```

```
        a,b=0,1
        i=1
        while i<=n:
            a,b=b,a+b
            i+=1
        return a
    elif n==0:
        return 0
    elif n==1:
        return 1
    else:
        print("n <0")
        return 0
# Вывод чисел Фибоначчи
for n in range(11):
    print("n= ", n, "Число Фибоначчи = ", fib(n))
```

Заметим, что в данном случае приведен далеко не самый «изящный» вариант алгоритма с целью лишь показать различные типы управляющих конструкций. В начале программы идет определение функции fib(n), вычисляющей число Фибоначчи для заданного номера n. Определение функций осуществляется с помощью инструкции def. Сразу заметим отличия в синтаксисе от C/C++. Во-первых, не указан тип передаваемого параметра функции и тип возвращаемого значения. Это, как уже указывалось, связано с тем, что Python - язык с динамической типизацией, так что тип определяемых переменных определяется по их значению. Во-вторых, мы не видим операторных фигурных скобок, открывающих и закрывающих тело функции и блоки операторов в управляющих структурах. Это еще одна особенность синтаксиса языка. Разработчики заметили, что в программах, написанных в хорошем стиле программирования, операторные скобки являются фактически избыточными, поскольку для хорошей читаемости кода программисты делают отступы в тексте программ в блоках кода управляющих конструкций и при определении функций. Скобки по-существу нужны только для компилятора, а читать текст удобнее, когда имеются отступы. В связи с этим, по-видимому, возникла идея отказаться от скобок совсем, но зато ввести жесткие правила на отступы, с помощью которых интерпретатор определял бы структуру выражений.

Программисту, привыкшему к синтаксису C/C++ поначалу отсутствие скобок кажется необычным, но затем, наоборот, более удобным, поскольку программы, написанные в одном стиле лучше читаются. Стиль в данном случае диктуется строгой регламентацией отступов: области кода, относящиеся к одной логической структуре должны иметь один и тот же отступ от начала строки. Количество пробелов в отступе строго не регламентируется, главное, чтобы оно для инструкций, относящихся к одной группе, было одинаковым. Рекомендуется использовать отступы в 4 пробела. Но это не жесткое правило, например, в Google принято использовать отступы в 2 пробела. Легко настроить на нужное число пробелов символ табуляции в текстовом редакторе и использовать его для создания отступов в тексте программы, хотя сам символ табуляции не рекомендуется использовать вместо пробелов, поскольку символ табуляции может иметь различные функции в разных системах. Поэтому лучше установить в настройках редактора опцию "Вставлять пробелы вместо табуляций».

В данной программе в функции fib(n) инструкции if, elif и else имеют отступ в одну табуляцию, код, находящийся внутри управляющих структур имеет по одной дополнительной табуляции для каждой вложенной структуры. После заголовков функций и управляющих структур ставится двоеточие. Конец тела функции или управляющей структуры интерпретатор определяет по возврату текста на предшествующий отступ.

Тип функции определяется по типу возвращаемого значения оператором return, которое должно быть одно, но может иметь структурированный тип, то есть, например, быть кортежем. Тело функции в нашем случае начинается с условного оператора if, разветвляющего ход вычислений в зависимости от величины передаваемого параметра. Операторы в теле каждого ветвления написаны с отступом от заголовка ветвления (ключевых слов if, elif, else).

Еще один элемент синтаксиса, на который следует обратить внимание - множественное присваивание перед и внутри цикла while. Сначала переменным a, b одновременно присваиваются значения 0 и 1, затем оно используется в цикле. При этом переменным в левой части присваиваются значения, получаемые в результате вычисления переменных в правой части от знака присваивания. Вычисления в правой части осуществляются слева направо.

Цикл while вычисляется пока выражение i<=n будет истинным. В Python, как и в C, любое ненулевое значение является истиной, ноль — ложь. В качестве условия может служить также строка, список — на самом деле любая последовательность. Последовательность с ненулевой

длиной является истиной, пустая — ложью. Проверка, использованная в примере, является простым сравнением. Стандартные операторы сравнения записываются так же, как в C: <, >, ==,

<= (меньше или равно), >= (больше или равно) и != (не равно).

Для увеличения итератора цикла используется оператор суммирования с присваиванием "+=". Инкрементный ("++") и декрементный ("--") операторы, имеющиеся в C/C++, в Python не работают.

В Python, в отличие от C нет функции "main". Основная часть программы пишется просто без отступа и определителя def. Она может включать инструкции для описания переменных, импорта внешних модулей и другие операторы.

Место, где переменной присваивается значение, определяет ее область видимости. Переменные, описанные в теле файла без отступа, имеют статус глобальных на уровне файла. При обращении к имени переменной внутри функции интерпретатор пытается последовательно отыскать ее в четырех областях видимости: а) в локальной внутри функции; б) локальной области любой объемлющей функции, если таковая имеется; в) в глобальной на уровне файла; г) среди встроенных имен языка. Поиск завершается как только будет найдено первое подходящее имя. Если необходимо при определении переменной внутри функции расширить ее область видимости до пределов файла, необходимо до ее определения использовать инструкцию global, например: global X (без присваивания значения).

В некоторых случаях имеет смысл создать функцию main(), которая будет являться точкой входа для интерпретатора и началом программы. В этом случае текст файла может вообще состоять только из определения функций. Для того чтобы интерпретатор "знал", что вычисления следует начинать именно с функции main() при запуске такого файла, в текст программы необходимо внести инструкцию:

```
if __name__ == "__main__" :    main()
```

Этот код проверяет, запущен ли данный файл как исполнимый, либо вызван другой программой в качестве внешнего модуля. В первом случае системной переменной (атрибуту файла) "__name__" интерпретатор автоматически присваивает значение "__main__". Тогда проверка условия даст результат "истина" и будет выполнена функция с именем main(). В противном случае (если файл был импортирован вызовом из другой функции) ничего не происходит - просто

объекты, определенные в нем, будут доступны в импортировавшем его модуле. Имя первой запускаемой функции не обязательно должно быть main. Можно передать управление при запуске любой функции.

В нашем случае основная часть программы состоит из одного цикла for, в котором выводятся с комментариями текущее значение n и результат вычисления функции fib(n). Следует обратить внимание на синтаксис записи цикла. Вместо явного описания начального значения переменной цикла, условия окончания и инкремента используется функция range() возвращающая список значений. Синтаксис этой функции достаточно гибкий, так что можно возвращать список значений с разным шагом. Вместо этой функции можно использовать в принципе любой список.

После окончания заголовка цикла ставится двоеточие, а тело цикла записывается с отступом, который выполняет функцию операторных скобок для интерпретатора. Функция print принимает список выражений, разделенных запятыми, оценивает их, пытается выполнить преобразование к строковому типу в тех случаях, когда это требуется, и передает их на стандартный выходной поток. Перед каждым выводимым выражением, кроме первого, вставляется дополнительно символ пробела. В конце добавляется символ перевода строки "\n" в том случае, если последний символ выражения не является запятой. В этом случае перевод строки не делается. Возможен также форматный вывод данных с помощью спецификаций, принятых в языке C для функции printf. То есть данная функция развивает далее идеи, заложенные в функцию cout в C++, дополняя ее новыми возможностями, и объединяя с функцией форматного вывода, принятой в языке C, и являющейся в некоторых случаях удобной.

Сохраните файл с именем "Fibonacci.py" и ознакомьтесь с работой программы.

Итак, мы знаем, как вывести результат. А как ввести данные в программу? Простейший способ ввода - передать данные в качестве параметров командной строки при запуске программы на исполнение. Для этой цели в модуле "sys" имеется функция, возвращающая список параметров, введенных в командной строке при запуске программы. Например, следующая программа выводит этот список:

```
import sys
def main():
   for par in sys.argv:
      i=sys.argv.index(par) # Вычисление индекса параметра par
```

```
        print("Индекс =", i, "Параметр:", par)
if __name__ == '__main__':  main()
```

В функции "main()", управление которой автоматически передается при запуске программы посредством управляющей конструкции if, с помощью цикла for вызывается и выводится на печать список аргументов командной строки, возвращаемый функцией sys.argv. Значениями итератора цикла являются сами элементы списка. Для того чтобы вычислить их индекс используется функция index, возвращающая индекс заданного элемента списка.

Сохраните файл с именем "sys_pars.py" и запустите его из терминала следующим образом:

```
python pt_2.py a1 a2 a3
```

В результате должно быть выведено:

```
Индекс = 0 Параметр: sys_pars.py
Индекс = 1 Параметр: a1
Индекс = 2 Параметр: a2
Индекс = 3 Параметр: a3
```

То есть первым параметром с индексом 0 является имя файла. Оно действительно является всегда первым параметром при вызове интерпретатора, поскольку передается ему в качестве параметра первым. Остальные параметры имеют индексы, начинающиеся с 1. Поэтому, чтобы получить, например, значение параметра "a2" необходимо было бы вызвать функцию как "sys.argv[2]".

Вводить данные в качестве параметра командной строки можно лишь в самых простых случаях. Для интерактивного ввода данных из терминала имеется функция input([prompt]), в качестве параметра при этом можно передавать приглашение ("prompt"), которое будет отображаться перед вводимой строкой. Эти функции возвращают введенный пользователем текст в виде строки. Если просто нажать «Enter» в приглашении, будет возвращена пустая строка.

Для файлового ввода и вывода используется функция open(name, [mode]), которая служит для открытия файла по имени "name" для чтения, записи или изменения. Режим может быть ’r’, если файл открывается только для чтения, ’w’ — только для записи (существующий файл будет перезаписан), и ’a’ — для дописывания в конец файла. В режиме ’r+’ файл открывается сразу для чтения и записи. Аргумент "mode" не является обязательным: если он опущен, подразумевается

’r’. В Windows файлы по умолчанию открываются в текстовом режиме — для того, чтобы открыть файл в двоичном режиме, необходимо к строке режима добавить ’b’. В целях переносимости программ рекомендуется это делать и в Linux.

Метод f.read([size]) считывает и возвращает некоторое количество данных из файла. Аргумент [size] не является обязательным. Если он опущен или отрицательный, будет считано все содержимое файла, в противном случае, будет считано не более size байт данных. Метод f.readline() считывает из файла одну строку. Возвращаемая строка всегда заканчивается символом новой строки (\n), за исключением последней строки файла, если файл не заканчивается символом новой строки. После того, как Вы закончили все операции с файлом, закройте файл с помощью f.close().

При работе с файлами удобно использовать конструкцию with, которая закрывает файл автоматически.

Строки легко могут быть записаны в файл и считаны из него. Числа требуют приложения небольших усилий, так как методы read() и readline() всегда возвращает строки, которые нужно обработать с помощью функций int() или float(). Разберите в качестве примера следующую программу:

```
L1=[44,45,46] # Создадим список из трех чисел
print(L1)
S1='' # Для записи его в файл необходимо сформировать строковую переменную
for n in L1:
    S1+=str(n)+' '
S1+='\n' # Последним символом переменной должен быть символ конца строки
with open('example.txt','w') as f:
    f.write(S1) # Записываем созданную переменную в файл example.txt
with open('example.txt','r') as f:
    L2 = f.readline().split() # считываем из файла и преобразуем ее в список
print(L2)
L2i=[int(x) for x in L2] # Преобразуем элементы списка в целые числа
print(L2i)
```

Задача сохранения и загрузки данных сильно усложняется, если необходимо сохранять более сложные типы данных, такие как списки, словари или экземпляры классов. Python предоставляет для этих целей стандартный модуль pickle. Этот модуль позволяет получить представление почти любого объекта в виде набора байтов (строки), одинакового для всех платформ. Такой процесс называют “консервированием” (pickling). Это представление (законсервированный объект) можно сохранить в файле или передать через сетевое соединение на другую машину. К считанному из файла или принятому на другой машине законсервированному объекту может быть применена операция восстановления (unpickling). Ниже приведен пример программы, в которой производится сохранение словаря в виде файла и считывание его:

```
import pickle
# Словарь, который нужно сохранить
data = {
    'Имя': 'Иван Петров',
    'возраст': 30,
    'город': 'Москва',
    'хобби': ['чтение', 'путешествия', 'телевизор']
}
# Сохранение словаря в файл
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)
print("Словарь сохранен в файл 'data.pkl'.")
# Загрузка словаря из файла
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)
print("Загруженный словарь:")
print(loaded_data)
```

Для решения задач математического моделирования в Python имеется пакет numpy, в которых определены новые типы данных и имеется большое количество различных функций. Для построения графиков функций можно использовать популярный программный пакет matplotlib. Вышеперечисленные пакеты не входят в состав стандартного дистрибутива Python и должны быть

предварительно установлены на компьютер из репозитория. Ниже приведен пример простой программы для построения графика функции с использованием библиотек numpy, matplotlib :

```
import numpy as np
import matplotlib.pylab as plt
R=2
t=np.linspace(0,np.pi*4,100)
x,y,x1,y1=[],[],[],[]
for i in range(100):
        x.append(pow(np.cos(t[i]),3))
        y.append(-2*abs(np.sin(t[i])))
        x1.append(R*(t[i]-np.sin(t[i]))/12.5-1)
        y1.append(R*(1-np.cos(t[i]))/5)
plt.plot(x,y,x1,y1)
plt.xlabel('x, x1')
plt.ylabel('y(x), y1(x1)')
plt.show()
```

Программа строит график двух функций y(x), y1(x1), одна из которых содержит параметр R. Для создания массива точек, равномерно заполняющих одномерный интервал, используется метод linspace(), определенный в библиотеке классов numpy. Сформированные массивы значений аргументов и функций передаются в качестве параметров метода plot библиотеки matplotlib.pylab. Методами xlabel(), ylabel формируются подписи к осям координат. Вывод графического окна осуществляется методом show(). Более подробно использование пакетов numpy и matplotlib будет рассмотрено в последующих главах.

Как и в других случаях синтаксис языка Python обеспечивает полиморфизм при передаче параметров в функциях. Например, определим следующую функцию:

```
>>> def product(a,b):
...     return a*b
```

В зависимости от типа передаваемых параметров результат действия функции будет различаться:

```
>>> product(2,3)
6
```

```
>>> product([1,2],3)
[1, 2, 1, 2, 1, 2]
>>> product('Hi! ',4)
'Hi! Hi! Hi! Hi! '
```

Следует отметить также возможности передавать в качестве параметров в функции произвольное число позиционных или именованных аргументов. В первом случае такой параметр при определении функции предваряется символом «*» (передаваемые параметры автоматически упаковываются в кортеж), во втором случае параметр указывается с символом «**» (передаваемые параметры автоматически упаковываются в словарь). При наличии таких параметров они должны указываться в списке параметров после позиционных и именованных аргументов. Ниже приведены примеры определения и вызовов функций с разным типом параметров:

```
>>> def f1(a,b,c=1):  # Определение функции с параметром по умолчанию
...     print(a,b,c)
>>> f1(2,b='two') # Вызов функции с передачей именованного параметра
2 two 1
>>> def f2(a,*b,**c):  # Определение функции с произвольным числом аргументов
...     print(a,b,c)
>>> f2(1,2,3,4,Ivanov=200,Petrov=400) # Вызов функции с разным типом аргументов
1 (2, 3, 4) {'Ivanov': 200, 'Petrov': 400}
```

Такого рода определения функций полезны, когда заранее не известно число параметров, которые необходимо передать при вызове функции. Например, следующая функция находит минимальное значение из произвольного числа переданных аргументов:

```
>>> def min_s(*arg):
...     return sorted(arg)[0]
```

Здесь использован метод сортировки. Тип передаваемых параметров может быть различным:

```
>>> min_s(2,3,5,1,7)
1
>>> min_s('mm','ba','qq')
'ba'
>>> min_s([3,5],[4,7],[1,4],[1,1])
```

[1, 1]

Поскольку имена функций также как и все остальные имена в Python являются ссылками, в качества параметров можно передавать и имена функций. Например, следующая функция «min_max» находит минимальное или максимальное значение переданной последовательности в зависимости от вида функции «test», переданной в качестве параметра:

```
# Поиск минимального и максимального значения в последовательности
def min_max(test,*args):
    res=args[0]
    for arg in args[1:]:
        if test(arg,res):
            res=arg
    return res
def less(x,y): return x<y
def more(x,y): return x>y
L=33,20,15,18,10,11
print ('Найдено минимальное:', min_max(less,*L))
print ('Найдено максимальное:', min_max(more,*L))
```

Обратите внимание, что при вызове функции «min_max» второй параметр передается с символом «*». В отличие от определения функции данном случае это означает «распаковку кортежа» в последовательность чисел.

1.3. Классы

Механизм классов в языке Python несколько отличается от C++ в том отношении, что является менее жестким по отношению к возможностям пользователя класса изменять определенные в нем объекты. Однако наиболее важные особенности классов полностью сохранены: механизм наследования допускает несколько базовых классов, производный класс может переопределить любые методы базовых классов, из метода можно вызывать метод с тем же именем базового класса. Объекты могут содержать произвольное количество собственных данных.

Фактически все типы данных в Python представляет собой объекты. Доступ к атрибутам объекта, как уже неоднократно указывалось, осуществляется по механизму квалифицированных имен:

```
object.attribute
```

Когда подобное выражение применяется к объекту, полученному с помощью инструкции «class», интерпретатор начинает поиск в дереве связанных объектов, который заканчивается, когда будет найдено первое появление атрибута «attribute».

Синтаксис инструкции «class» выглядит следующим образом:

```
class <имя_класса>(класс1, класс2, ...):
    # определения данных и методов
```

Здесь объекты «класс1» и «класс2» представляют собой родительские классы, так что новый объект при его создании наследует их атрибуты. Порядок поиска атрибутов в родительских классах (если они имеются) соответствует порядку их перечисления слева направо.

Каждый раз, когда вызывается класс, он создает новый объект экземпляра класса. Этот объект наследует все его атрибуты, образующие древовидную структуру от родительских классов. Атрибуты присоединяются к классам при помощи инструкций присваивания внутри определения класса. Методы класса (функции определенные внутри класса) создаются с помощью инструкции «def». Специальный аргумент «self», указываемый в качестве первого аргумента функций при их определении и в качестве имени объекта при определении переменных, обеспечивает присоединение атрибутов экземплярам классов. Этот аргумент фактически представляет собой ссылку на обрабатываемый экземпляр.

Рассмотрим операции с классами на примере программы с определением простейшего класса:

```
class FirstClass:
   "Simple class example"
   a=1
   b=2
   def calc_res(self,c):
      self.res_a=(self.a+c)/2.0
      self.res_b=(self.b+c)/2.0
   def disp_res(self):
      print ('(a+c)/2 =',self.res_a,'(b+c)/2 =',self.res_b)
```

```
x=FirstClass()
c=4
x.calc_res(c)
print ('a=', x.a,'b=', x.b,'res_a=',x.res_a,'res_b=',x.res_b)
x.disp_res()
```

Сохраните данную программу в файле «simp_class_1.py» и импортируйте ее в интерактивной оболочке с помощью инструкции

```
>>> import simp_class_1 as cl_1
a= 1 b= 2 res_a= 2.5 res_b= 3.0
(a+c)/2 = 2.5 (b+c)/2 = 3.0
```

При импорте автоматически производится загрузка модуля в память и вычисления, так что в результате отображаются результаты инструкций print и x.disp_res().

Классы поддерживают два вида операций: доступ к атрибутам и создание экземпляра класса. Именами атрибутов класса являются все имена, определенные при его описании. Например, мы можем получить доступ к переменным:

```
>>> cl_1.FirstClass.a
1
>>> cl_1.FirstClass.b
2
>>>
```

Первым при вызове атрибутов указывается имя модуля, в котором определен класс, далее имя класса и затем имя атрибута. Строка документации, определенная как текстовая переменная после заголовка класса также представляет собой атрибут класса, имеющий специальное имя __doc__:

```
>>> cl_1.FirstClass.__doc__
'Simple class example'
```

Создание экземпляра класса использует синтаксис вызова функций. Так в инструкции x=FirstClass() создается экземпляр класса. Вызов методов класса для этого экземпляра осуществляется инструкциями x.calc_res(c) и x.disp_res(). Мы можем создавать экземпляры класса в интерактивной оболочке и вызывать для них соответствующие атрибуты и методы:

```
>>> c=6
>>> y=cl_1.FirstClass()
>>> y.calc_res(c)
>>> y.res_a
3.5
>>> y.disp_res()
(a+c)/2 = 3.5 (b+c)/2 = 4.0
```

Заметим, что атрибуты, определяемые в методах класса автоматически не присоединяются к экземплярам класса при их создании.

```
>>> z=cl_1.FirstClass()
>>> z.a
1
>>> z.res_a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: FirstClass instance has no attribute 'res_a'
```

Для этого соответствующий метод должен быть предварительно вызван, так как это было сделано в случае с экземпляром «y». Если необходимо, чтобы некоторые атрибуты автоматически присоединялись к экземплярам классов при их создании, необходимо использовать в определении класса специальный метод «__init__», например:

```
>>> class Vector:
...     def __init__(self, coord_X,coord_Y,coord_Z):
...             self.x=coord_X
...             self.y=coord_Y
...             self.z=coord_Z
>>> v=Vector(1,2,3)
>>> v.x,v.y,v.z
(1, 2, 3)
```

Метод __init__ автоматически вызывается каждый раз при создании экземпляра класса. Новый экземпляр передается этому методу в качестве первого аргумента self, а затем передаются последующие аргументы (если они имеются).

Итак, в модели объектно-ориентированного программирования Python существуют две разновидности объектов: объекты классов и объекты экземпляров классов. Первые служат для производства вторых. Объекты классов создаются инструкциями, и в этом смысле они напоминают модули. Так же как и в случае модулей при создании объектов классов создается ассоциированное с ними пространство имен. Объекты экземпляров создаются вызовами классов. В этом состоит принципиальное отличие классов от модулей: модуль всегда имеется в единственном экземпляре, тогда как путем многократного вызова класса можно создавать множество его экземпляров.

Инструкция «class» создает объект класса и присваивает ему имя. Внутри инструкции «class» операции присваивания на верхнем уровне (не вложенные в инструкции «def») создают атрибуты объекта класса (данные). После этого они становятся доступными по механизму квалифицированных имен: «object.name». Они хранят информацию и описывают поведение, которым обладают все экземпляры класса. Инструкции «def», вложенные в инструкцию «class» создают методы, которые обрабатывают экземпляры.

Вызов объекта класса как функции создает объект экземпляра. Каждый объект экземпляра наследует атрибуты класса и приобретает собственное пространство имен, например:

```
>>> class C1: # Создаем объект  класса C1
...     a=1
>>> x=C1() # Создаем экземпляр класса C1
>>> x.a  # Доступ экземпляра к атрибуту класса
1
>>> x.b=2 # Создание у экземпляра нового атрибута
>>> y=C1() # Создаем еще экземпляр класса C1
>>> y.a
1
>>> y.b # Пространства имен разных экземпляров не совпадают
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: C1 instance has no attribute 'b'
```

Операции присваивания значений через ссылку «self» в методах класса создают атрибуты в каждом отдельном экземпляре. Методы класса через ссылку «self» в первом аргументе обеспечивают создание и изменение данных в обрабатываемом экземпляре, а не классе.

Механизм наследования позволяет в дочернем классе переопределять (и добавлять новые) атрибуты родительского (базового) класса и использовать его методы:

```
>>> class SecondClass(cl_1.FirstClass):
...     a=3
...     b=4
>>> x=SecondClass()
>>> x.calc_res(4)
>>> x.disp_res()
(a+c)/2 = 3.5 (b+c)/2 = 4.0
```

Механизм сокрытия данных в Python не столь жесткий как в C++. В большей степени он основан на соглашениях, нежели на синтаксисе. Так, одиночное подчеркивание в начале имени атрибута указывает на то, что он не входит в общедоступный интерфейс. Однако, это не более чем соглашение, которое как бы говорит программисту: «этот атрибут для внутреннего использования». При этом атрибут остается доступным наравне с другими. Двойное подчеркивание работает как указание на то, что атрибут приватный. Он уже не доступен при прямом обращении, однако, он все же доступен под другим именем, которое получается присоединением перед именем атрибута имени класса с префиксом одиночного подчеркивания:

```
>>> class X:
...     x=0
...     _x=1
...     __x=2
...
>>> y=X
>>> y.x
0
>>> y._x
```

```
1
>>> y.__x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: class X has no attribute '__x'
>>> y._X__x
2
```

В заключение приведем пример, резюмирующий особенности области видимости переменных в языке Python. Рассмотрим следующую программу «names_vision.py»:

```
X=111       # Глобальное имя (атрибут) в модуле
def f1():
   print( 'in f1 X=', X)      # Обращение к глобальному имени «X»
def f2():
   X=222   # Локальная переменная в функции «f2» скрывает глобальное имя
   print( 'in f2 X=', X)
class C:
   X=333   # Атрибут класса C
   def f3(self):
      X=444   # Локальная переменная в методе f3 класса C
      self.X=555 # Атрибут экземпляра класса C
```

В этом файле пять раз определяется переменная X. Однако, поскольку каждый раз присваивание производится в различных областях видимости, все пять имен представляют ссылки на совершенно различные объекты. Присваивание «X=111» осуществляется на уровне модуля и приводит к определению глобальной переменной на уровне модуля. Присваивание «X=222» осуществляется на уровне функции «f2» и приводит к определению локальной переменной, которая в области видимости данной функции замещает глобальную переменную с тем же именем. Присваивание «X=333» приводит к созданию атрибута класса «C», видимость этой переменной ограничена областью видимости класса. Присваивание «X=444» в методе «f3» класса создает локальную переменную функции «f3». Ее особенностью является то, что поскольку она определена в методе класса не как атрибут экземпляра класса, область ее видимости ограничена только данной функцией и временем ее выполнения. Доступ к ней извне невозможен. Точно также невозможен и

доступ извне к значению переменной «X», определенной в функции «f2». Присваивание «self.X=555» в методе «f3» класса создает локальную переменную экземпляра класса. Доступ к ней возможен как к атрибуту экземпляра класса.

Для иллюстрации этих особенностей загрузите модуль «names_vision.py» в интерактивной оболочке:

```
>>> import names_vision as n_v
```

Определим глобальную переменную «X» на уровне интерактивной оболочки

```
>>> X=666
>>> X
666
```

Глобальная переменная «X» на уровне модуля при этом не изменилась:

```
>>> n_v.X
111
```

Вызов глобальной переменной «X» на уровне модуля из функции «f1»:

```
>>> n_v.f1()
in f1 X= 111
```

Выведем значение локальной переменной «X» в функции «f2»:

```
>>> n_v.f2()
in f2 X= 222
```

Доступ извне к данной переменной невозможен. Выведем значение переменной «X» в классе «C» (атрибут класса):

```
>>> n_v.C.X
333
```

Создадим экземпляр класса «C»:

```
>>> exm=n_v.C()
```

Экземпляр наследует значение переменной «X» в классе «C»:

```
>>> exm.X
```

```
333
```

Вызовем метод «f3» для экземпляра класса.

```
>>> exm.f3()
```

Тем самым мы присоединили атрибут «X=555» к экземпляру:

```
>>> exm.X
555
```

Доступ извне к локальной переменной «X=444», определенной в функции «f3» невозможен. Глобальная переменная «X» при всех этих вызовах не изменилась:

```
>>> X
666
```

Стоит отметить, что пространства имен классов и их экземпляров, как и модулей, реализованы в форме словарей. Эти словари доступны с помощью специального атрибута «__dict__», так что значения атрибутов классов и экземпляров могут быть выведены также с использованием этих словарей:

```
>>> n_v.C.__dict__
mappingproxy({'__module__': 'names_vision', 'X': 333, 'f3': <function C.f3 at 0x775f703f4160>, '__dict__': <attribute '__dict__' of 'C' objects>, '__weakref__': <attribute '__weakref__' of 'C' objects>, '__doc__': None})
>>> exm.__dict__
{'X': 555}
>>> exm.__dict__['X']
555
```

Развитие языка Python осуществляется в направлении все большего использования классов. В частности это относится к механизмам обработки исключений (инструкции "try/except/finally", "raise","assert") и использованию контекстных менеджеров (инструкция "with/as").

Рассмотрим более подробно обработку исключений. Исключения в Python возбуждаются автоматически, когда программный код допускает ошибку, кроме того, они могут возбуждаться и перехватываться самим программным кодом при помощи специальных инструкций. При этом исключения, определяемые программой должны быть экземплярами классов. Программная обработка исключений применяется не только для отслеживания ошибок, но также и обработки

особых ситуаций, управления потоком выполнения и уведомления пользователя о наступлении некоторых событий.

Конструкция "try/except" имеет следующий синтаксис:

```
try:
    <основной блок>
except: <Исключение группы 1> [as <Переменная 1>]
    <блок исключения 1>
...
[except: <Исключение группы N> [as <Переменная N>]
    <блок исключения N>]
[else:
    <дополнительный блок>]
[finally:
    <заключитеный блок>]
```

Эта конструкция должна содержать хотя бы один блок "except", а блоки "else" и "finally" являются необязательными. Дополнительный блок выполняется, только если основной блок завершается обычным способом, и не выполняется в случае возникновения исключения. Если блок "finally" присутствует, то он выполняется всегда и в последнюю очередь. Каждое исключение в предложениях "except" должно быть единственным именем класса исключения или кортежем имен классов исключений в круглых скобках. Часть "as <Переменная>" в каждой группе является необязательной. В случае ее использования соответствующей переменной присваивается ссылка на имя группы классов исключений, которую можно использовать в блоке исключения.

Если при работе программы исключение возникает в основном блоке, интерпретатор поочередно начинает проверять каждое предложение "except". В случае отнесения исключения к какой-либо из групп, начинает выполняться блок, соответствующий данной группе исключений. При этом к соответствующим относятся также и все исключения, входящие в зарегистрированные подклассы каждого из перечисленных в группе классов исключений. Все встроенные классы исключений являются подклассами базового класса "Exception". К их числу относятся классы: "ArithmeticError", "EnvironmentError" (подклассы: "IOError", "OSError"), "EOFError", "LookupError" (подклассы: "IndexError", "KeyError"), "ValueError". Исключения, не

перехваченные предложениями "except" распространяются до самого верхнего уровня процесса (если, например, инструкция "try" является вложенной) и запускают логику обработки исключений по умолчанию (то есть интерпретатор аварийно останавливает программу и выдает сообщение об ошибке).

Пользователь может создавать собственные классы исключений, так что рассматриваемая конструкция имеет более широкую область применимости в отношении управления ходом процесса, нежели просто обработка ошибок.

Очень часто конструкция "try/except/finally" используется для обработки ошибок, возникающих при работе с файлами. Например, следующий фрагмент кода считывает данные из текстового файла построчно и формирует на их основе список строк с удаленными ведущими и концевыми символами пробела:

```
def read_strings(filename):
    lines=[]
    fl=None
    try:
        fl=open(filename, encoding = 'utf-8')
        for line in fl:
            if line.strip():
                lines.append(line)
    except (IOError,OSError) as err:
        print(err)
    finally:
        if fl is not None:
            fl.close()
    return lines
```

Изначально функция создает пустой список с именем "lines" и записывает в переменную "fl" значение "None", для того чтобы в случае не успешного открытия файла было возбуждено исключение. В этом случае обработчик ошибок выведет сообщение об ошибке и функция вернет пустой список. Вне зависимости от того, было возбуждено исключение или нет, инструкция "finally" обеспечивает закрытие файла, если он был открыт.

В качестве альтернативы использования конструкции "try/finally" может быть использована инструкция "with/as". Она также позволяет выполнить завершающие действия независимо от того, возникло ли исключение на этапе выполнения основного действия, но кроме того, поддерживает более богатый возможностями протокол, позволяющий определять как предварительные, так и заключительные действия для заданного блока программного кода. Инструкция "with/as" предназначена для работы с менеджерами контекста, ее синтаксис выглядит следующим образом:

```
with <выражение> [as переменная]:
    <основной блок >
```

Здесь предполагается, что выражение возвращает объект, поддерживающий протокол менеджера контекста. Этот объект может возвращать значение, которое будет присвоено переменной, если в инструкции присутствует необязательное ключевое слово "as", или отброшено, в противном случае. Объект, возвращаемый выражением, может затем выполнять предварительные действия перед тем, как будет запущен блок, а также завершающие действия, после того как блок будет выполнен, независимо от того, было ли возбуждено исключение при его выполнении. Таким образом, для контроля за открытием файлов вместо рассмотренной выше конструкции "try" может быть использована конструкция:

```
with open(filename) as fl:
    for line in fl:
        if line.strip():
            lines.append(line)
... <остальной код программы>
```

Здесь вызываемая функция "open" возвращает объект файла, поддерживающий протокол менеджера контекста, который присваивается переменной "fl". К данной переменной применимы обычные средства, принятые для работы с файлами. После того, как инструкция "with" начнет выполнение, механизм управления контекстом гарантирует, что что объект файла, на который ссылается переменная "fl", будет закрыт автоматически, даже если в цикле "for" произойдет исключение. Помимо использования встроенных менеджеров контекста имеется возможность создавать собственные классы менеджеров контекста, с помощью которых можно управлять ходом выполнения процесса.

1.4. Работа с базами данных

Язык Python включает в себя интерфейс для работы с базами данных SQL и распространяется с поддержкой базы данных SQLite 3. Кроме этого поддерживается формат баз данных DBM (Data Base Manager), в котором данные хранятся в виде произвольного числа элементов "ключ - значение". Базы данных DBM работают по принципу словарей в языке Python, при этом ключи и значения всегда являются объектами типа "bytes". Модуль "shelve" предоставляет удобный интерфейс DBM, позволяя использовать строковые ключи и любые поддающиеся консервированию объекты в качестве значений. Данный модуль представляет собой обертку вокруг DBM, позволяя взаимодействовать с базой данных как с обычным словарем. Ниже приведен текст программы для создания базы данных геометрических объектов в формате DBM.

```
import shelve
def typed_inp(prompt="Введите данные: ",type_of=int, maxcount=3):
    '''Typed input'''
    entered_ls=[]
    count=0
    while count < maxcount:
        try:
            line=input(prompt)
            if line:
                entered = type_of(line)
                entered_ls.append(entered)
                count += 1
            else:
                break
        except ValueError:
            err='Ошибка ввода данных'
            print(err)
            continue
        except EOFError:
            break
    return entered_ls
```

```
    def choose_1(prompt='Выберите вариант:',type_of=str,allowed=lambda x: x in '_', right_choise_msg='Принимается',wrong_choise_msg='Не принимается',maxcount=3):
        '''Typed choose'''
        count=0
        while count < maxcount:
            try:
                choise=type_of(input(prompt))
                if allowed(choise):
                    print( right_choise_msg)
                    return choise
                else:
                    print( wrong_choise_msg )
                    count += 1
            except ValueError:
                err='Ошибка ввода данных'
                print(err)
                count += 1
                continue
    def add_rec(db):
        points={}
        lines={}
        arcs={}
        title=input('Введите название записи: ')
        if not title:
            print( 'Ошибка ввода данных' )
            return
        if title in db.keys():
            print( 'Запись с таким названием уже имеется в базе' )
            return
        add_continue='yes'
```

```
        while add_continue in ('yes','y','OK','да','продолжить'):
            primitive=choose_1(prompt='Выберите тип примитива (точка - т, линия - л, дуга - д)
: ',
                type_of=str,allowed=lambda x: x in 'тлд', right_choise_msg='Принимается',
                wrong_choise_msg='Не принимается',maxcount=3)
            if primitive=='т':
                point_name=input('Введите название точки: ')
                if not point_name:
                    print( 'Ошибка ввода названия' )
                    return
                if point_name in points.keys():
                    print( 'Точка с таким названием уже существует' )
                    return
                        poin_coords=typed_inp(prompt="Введите  координату:  ",type_of=float,
maxcount=3)
                if len(poin_coords)==3:
                    points[point_name]=poin_coords
            if primitive=='л':
                print( 'Отрезок задается названием и выбором двух точек из имеющихся.' )
                line_name=input('Введите название отрезка: ')
                if not line_name:
                    print( 'Ошибка ввода названия' )
                    return
                if line_name in lines.keys():
                    print( 'Отрезок с таким названием уже существует' )
                    return
                print( 'Имеются точки:' )
                for key in points.keys():
                    print( key, points[key] )
                point_name_1=choose_1(prompt='Введите название первой точки отрезка: ',
                                                    type_of=str,allowed=lambda  x:  x  in  points.keys(),
right_choise_msg='Принимается',
```

```
                wrong_choise_msg='Не принимается',maxcount=1)
            lk=list(points.keys())
            lk.pop(lk.index(point_name_1))
            point_name_2=choose_1(prompt='Введите название второй точки отрезка: ',
                type_of=str,allowed=lambda x: x in lk, right_choise_msg='Принимается',
                wrong_choise_msg='Не принимается',maxcount=1)
            if point_name_1 and point_name_2:
                lines[line_name]=[point_name_1,point_name_2]
        if primitive=='д':
            print( 'Дуга задается названием, выбором точки центра, радиусом, начальным и конечным углами.' )
            arc_name=input('Введите название дуги: ')
            if not arc_name:
                print( 'Ошибка ввода названия' )
                return
            if arc_name in arcs.keys():
                print( 'Дуга с таким названием уже существует' )
                return
            print( 'Имеются точки:' )
            for key in points.keys():
                print( key, points[key] )
            point_name_r=choose_1(prompt='Введите название точки центра дуги: ',
                type_of=str,allowed=lambda x: x in points.keys(), right_choise_msg='Принимается',
                wrong_choise_msg='Не принимается',maxcount=1)
            arc_attr=[point_name_r]
            arc_rad=choose_1(prompt='Введите радиус дуги: ',
                type_of=float,allowed=lambda x: x > 0.0, right_choise_msg='Принимается',
                wrong_choise_msg='Не принимается',maxcount=1)
            print( 'Ввод углов начала и конца дуги' )
                arc_angls=typed_inp(prompt="Введите угол в градусах: ",type_of=float, maxcount=2)
```

```
            arc_attr.append(arc_rad)
            arc_attr.extend(arc_angls)
            arcs[arc_name]=arc_attr
        add_continue=input('Продолжить ввод примитивов? ')
    db[title]=(points,lines,arcs)
    db.sync()
def del_rec(db):
    title=input('Введите название:')
    if not title:
        print( 'Ошибка ввода данных' )
        return
    if title not in db.keys():
        print( 'Запись с таким названием отсутствует в базе' )
        return
    conf=input('Запись '+title+' будет удалена? (yes,no): ')
    if conf=='yes':
        del db[title]
        db.sync()
        return
    else:
        return

db=None
try:
    filename=input('Введите имя файла базы данных: ')
    db=shelve.open(filename)
    add_continue='yes'
    while add_continue in ('yes','y','OK','да','продолжить'):
        print( 'Имеются записи:')
        for key in db.keys():
            print( key, db[key], '\n')
```

```
                proc_db=choose_1(prompt='Введите операцию с записями (добавить - add, удалить - del): ',
                                                type_of=str,allowed=lambda x: x in ('add','del'), right_choise_msg='Принимается',
                    wrong_choise_msg='Не принимается',maxcount=1)
            if proc_db=='add':
                add_rec(db)
            if proc_db=='del':
                del_rec(db)
            add_continue=input('Продолжить работу с базой? ')
        finally:
            if db is not None:
                db.close()
```

База данных представляет собой словарь, ключи которого соответствуют записям, в которых хранятся наборы геометрических примитивов: точек, линий и дуг окружностей. Каждый набор оформлен в виде словаря, ключами которого являются наименования примитивов, а значениями - их параметры. Для точек параметром является список их координат; для линий - список из двух элементов: наименований точек, составляющих концы отрезка; для дуг список состоит из наименования точки центра дуги и трех чисел: радиуса, угла начала и угла конца дуги.

Программа начинается с двух вспомогательных функций "typed_inp" и "choose_1", которые позволяют организовывать интерфейс с пользователем при вводе информации с терминала. Первая из этих функций позволяет формировать списки данных заданного типа, вторая функция позволяет осуществлять ввод данных с проверкой на соответствие заданным требованием. Это достигается путем передачи в качестве параметра "allowed" выражений в виде "лямбда-функций", которые осуществляют проверку вводимых данных. Вспомогательные функции далее используются в функции добавлении записей в базу данных ("add_rec") и в основной части программы.

В основной части с помощью метода "shelve.open(filename)" открывается (или создается при его отсутствии) файл базы данных и в цикле "for key in db.keys()" отображаются имеющиеся в ней записи. Далее с помощью функции "choose_1" пользователю предлагается выбрать вариант

работы с базой - добавить запись с помощью функции "add_rec" или удалить запись, используя функцию " del_rec".

В функции "add_rec" пользователю предлагается ввести название записи, являющейся ключом словаря записей базы данных и заполнить кортеж из трех словарей "(points,lines,arcs)", являющийся значением для данного ключа. Элементами кортежа являются, в свою очередь, словари, отвечающие наборам каждого из видов примитивов: точек - "points", линий - "lines" и дуг - "arcs". Поскольку для того чтобы определить линию или дугу необходимо уже иметь необходимые точки (для линии - начало и конец отрезка, для дуги - точку центра), то заполнение словарей начинается с создания двух точек в цикле "while len(points.keys())<2". Затем посредством инструкции "primitive=choose_1(...)" пользователь может выбрать продолжение создания точек или же перейти к созданию линий и дуг. Условие окончания заполнения записи проверяется в цикле "while add_continue in ('yes','y','OK','да','продолжить'):". При окончании ввода примитивов создается новый элемент словаря "db[title]=(points,lines,arcs)" и производится синхронизация базы данных "db.sync()".

В функции "del_rec" пользователю предлагается ввести название записи в базе данных, и если соответствующий ключ в словаре имеется (проверка условия "if title not in db.keys()"), то данный элемент словаря будет удален.

Ознакомьтесь с работой данной программы, проанализируйте, какие еще функции по управлению данными могут быть добавлены.

Преимущества использования модуля "shelve" заключаются в возможности хранить в базе данных любой объект, подвергающийся упаковке с помощью модуля "pickle", то есть практически любой тип данных (как встроенный, так и создаваемый пользователем), поддерживаемый в Python. Вместе с тем, недостаток, являющийся следствием этой универсальности, состоит в том, на программиста возлагается задача самостоятельного создания функций управления данными, как, например, это было реализовано в рассмотренной выше программе. В том случае, если данные имеют простые стандартные форматы, поддерживаемые общепринятыми СУБД, удобнее работать с модулем базы данных SQLite 3. В этом случае для управления данными в распоряжении пользователя имеются широкие возможности, предоставляемые SQL.

Основными объектами, поддерживаемыми модулем "sqlite3" являются объект соединения и объект курсора. Объект соединения имеет синтаксис

db=sqlite3.connect(filename)

Он открывает или создает (при его отсутствии) файл "filename" базы данных SQLite 3. Основные методы объекта соединения - это создание объекта курсора, посредством которого осуществляются запросы к базе ("db.cursor()"), подтверждение транзакций в базе данных ("db.commit") и закрытие соединения ("db.close").

Объект курсора, создаваемый инструкцией "c=db.cursor", поддерживает следующие основные методы:

"c.execute(sql,params)" - выполняет SQL-запрос, содержащийся в строке "sql", замещая каждый символ-заполнитель соответствующим параметром из последовательности или отображения "params", если таковые имеются;

"c.fetchone()" - возвращает следующую строку из набора результатов, полученных в результате запроса;

"c.fetchall()" - возвращает все строки, которые еще не были извлечены;

"c.close()" - закрывает курсор "c" (эта операция осуществляется автоматически при выходе из области видимости курсора).

Кроме того,имеются методы "executemany" и "fetchmany" для работы с последовательностями.

Ниже приводится примерный текст программы для работы с базой данных результатов испытаний. В качестве объектов испытаний могут выступать, например, некоторые материалы или изделия. База состоит из трех таблиц:

1. Таблица объектов "Names", имеющая поля: "Id" (индекс записи - ключевое поле с автозаполнением), "Class" (текстовое поле для описания класса испытываемых объектов), "Name" (текстовое поле для названия испытываемого объекта);

2. Таблица описания методов испытаний "Prop_Descr" с полями: "Id" (индекс записи - ключевое поле с автозаполнением), "Prop_Name" (текстовое поле для названия испытания), "Prop_Descript" (текстовое поле для описания метода испытания);

3. Таблица результатов испытаний "Prop_Vals", содержащая, помимо ключевого индекса, поля: "Prop_Val" (для результатов испытания), "Prop_Comm" (для комментариев к результатам), "Prop_Descr_id" (индекс записи в таблице "Prop_Descr", относящейся к данному испытанию), Names_id (индекс записи в таблице "Names", относящейся к испытываемому материалу).

Следует заметить, что в отличие от многих других баз данных SQLite 3 подобно языку Python поддерживает динамическую типизацию данных, так что в действительности тип данных содержащихся в конкретном поле может быть любым. Кроме того, SQLite 3 автоматически поддерживает реляционные связи между таблицами, что достигается указанием спецификации "FOREIGN KEY" для зависимого поля.

```
import sqlite3
import os
def choose_1(prompt='Выбрать  запись из имеющихся - выб, создать новую - доб или нет :',
        type_of=str,allowed=lambda x: x in ['доб','выб','нет'],
        right_choise_msg='',wrong_choise_msg='Не принимается',maxcount=3):
    '''Typed choose'''
    count=0
    while count < maxcount:
        try:
            choise=type_of(input(prompt))
            if allowed(choise):
                print(right_choise_msg)
                return choise
            else:
                print( wrong_choise_msg )
                count += 1
        except ValueError:
            err='Ошибка ввода данных'
            print(err)
            count += 1
            continue
```

```
    def choose_file(chose_dir='./',file_type=''):
        ls_curdir=os.listdir(chose_dir)
        if chose_dir=='./':
            chose_dir='текущем каталоге'
        print( 'В ' +chose_dir+  ' имеются файлы: \n', ls_curdir )
        filename=input('Введите имя файла '+file_type+': ')
        if filename:
            create = not os.path.exists(filename)
            if create:
                ans=choose_1(prompt='Создать новый файл '+filename+'(да,нет) ?: ',
                        type_of=str,allowed=lambda x: x in['да','нет'],
                                                right_choise_msg='Принимается',wrong_choise_msg='Не
принимается',maxcount=3)
                if ans=='да':
                    return [filename,'create']
            else:
                return [filename,'exists']


    def view_table(db, table_name, field_names='*', condition=''):
        cur = db.cursor()
        sql_stm = "SELECT "+field_names+" FROM "+table_name+condition
        cur.execute(sql_stm)
        rows = cur.fetchall()
        num_rows=len(rows)
        i=0
        while True:
            print( rows[i][0],rows[i][1],rows[i][2] )
                ans=choose_1(prompt='Выбрать текущую запись: да; вперед: +кол-во; назад: -
кол-во;удалить запись - уд; отказ: нет ?: ', type_of=str, allowed=lambda x: x in ['да','нет','уд'] or
(len(x)>1 and (x[0] in '+-' and x[1:]).isdecimal()),right_choise_msg='',wrong_choise_msg='Не
принимается',maxcount=3)
```

```
            if ans=='да':
                return rows[i]
            if ans=='уд':
                conf_ans=choose_1(prompt='Подтверждаете удаление текущей записи? да или
нет :',
                type_of=str,allowed=lambda x: x in ['да','нет'],
                right_choise_msg='',wrong_choise_msg='',maxcount=1)
                if conf_ans=='да':
                    sql_stm ="DELETE FROM " +table_name+ " WHERE Id="+str(rows[i][0])
                    cur.execute(sql_stm)
                    sql_stm = "SELECT "+field_names+" FROM "+table_name+condition
                    cur.execute(sql_stm)
                    rows = cur.fetchall()
                    num_rows=len(rows)
            if ans=='нет':
                return
            if ans[0]=='+':
                i +=int(ans[1:])
            if ans[0]=='-':
                i -=int(ans[1:])
            if i<0: i=num_rows-1
            if i>num_rows-1: i=0


        def add_rec(db, table_name, field_names, field_vals):
            cur = db.cursor()
            sql_stm = "INSERT INTO "+table_name+" "+field_names
            num_fields = len(field_vals)
            val = " VALUES ("+"?,"*(num_fields-1)+"?) "
            sql_stm += val
            cur.execute(sql_stm,field_vals)
```

```
        db.commit()
        sql_stm = "SELECT * FROM "+table_name
        cur.execute(sql_stm)
        rows = cur.fetchall()
        return rows[-1]


    filename=choose_file(chose_dir='./',file_type='базы данных')
    if filename:
        with sqlite3.connect(filename[0]) as db:
            if filename[1]=='create':
                cur = db.cursor()
                            cur.execute("CREATE TABLE Names(Id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, Class TEXT NOT NULL, Name TEXT UNIQUE NOT NULL)")
                        cur.execute("CREATE TABLE Prop_Descr(Id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, Prop_Name TEXT UNIQUE NOT NULL, Prop_Descript TEXT)")
                         cur.execute("CREATE TABLE Prop_Vals(Id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, Prop_Val NUMERIC NOT NULL,Prop_Comm TEXT, Prop_Descr_id INT NOT NULL,  Names_id INT NOT NULL, FOREIGN KEY (Names_id) REFERENCES Names (Id) )")
                cur.execute("PRAGMA foreign_keys =ON")
                db.commit()
            elif filename[1]=='exists':
                cur=db.cursor()
                cur.execute("PRAGMA foreign_keys =ON")
            add_continue='yes'
            while add_continue in ('yes','y','OK','да','продолжить'):
                print( 'Работа с таблицей Объекты' )
                try:
                    table_name='Names'
                    ans=choose_1(allowed=lambda x: x in ['доб','выб'])
                    if ans=='выб':
```

```
                rec_Name=view_table(db, table_name)
            elif ans=='доб':
                field_names='(Class, Name)'
                Class_v=input('Ввведите класс объекта: ')
                Name_v=input('Ввведите название объекта: ')
                rec_Name=add_rec(db, table_name, field_names, (Class_v,Name_v))
            elif rec_Name ==[] or ans=='нет':
                print( 'Объект не выбран' )
                break
        except NameError:
            print( 'Ошибка ввода' )
            break
        print( 'Работа с таблицей Свойства' )
        try:
            table_name='Prop_Descr'
            ans=choose_1(allowed=lambda x: x in ['доб','выб'])
            if ans=='выб':
                rec_Prop=view_table(db, table_name)
            elif ans=='доб':
                field_names='(Prop_Name, Prop_Descript)'
                Prop_Name_v=input('Введите название свойства объекта: ')
                Prop_Descript_v=input('Ввведите описание свойства: ')
                                        rec_Prop=add_rec(db,  table_name,  field_names,
(Prop_Name_v,Prop_Descript_v))
            elif rec_Prop==[] or ans=='нет':
                print( 'Свойство не выбрано' )
                break
        except NameError:
            print( 'Ошибка ввода' )
            break
        if  rec_Name  and rec_Prop:
```

```
            print( 'Работа с таблицей Данные' )
            try:
                table_name='Prop_Vals'
                ans=choose_1(allowed=lambda x: x in ['доб','выб'])
                if ans=='выб':
                    rec_Val=view_table(db, table_name)
                elif ans=='доб':
                    Prop_Val_v=input('Введите значение выбранного свойства объекта: ')
                        Prop_Comm_v=input('Введите примечание к введенному значению (если имеется): ')
                    Names_id_v=rec_Name[0]
                    Prop_Descr_id_v=rec_Prop[0]
                    field_names='(Prop_Val, Prop_Comm, Names_id, Prop_Descr_id)'
                                rec_Val=add_rec(db, table_name, field_names, (Prop_Val_v, Prop_Comm_v, Names_id_v, Prop_Descr_id_v))
                if  rec_Val==[]:
                    print( 'Данные не записаны' )
            except NameError:
                print( 'Ошибка ввода' )
                break
        add_continue=input('Продолжить работу с базой? ')
```

В качестве первой вспомогательной функции используется функция "choose_1" из предыдущего примера, отличающаяся лишь значением параметров по умолчанию. Вторая вспомогательная функция "choose_file" предназначена для выбора файла базы данных при начале работы с программой или создания его, если он отсутствует в текущем каталоге. Функция возвращает список из ссылки на файл и пояснение, был ли он открыт как новый ("create") или имеющийся ("exists"). Вспомогательная функция "view_table" предназначена для просмотра, выбора и удаления записей из имеющейся таблицы. В качестве параметров данная функция принимает ссылку на открытый файл базы данных ("db"), имя таблицы ("table_name"), текстовую переменную для описания требуемых полей ("field_names='*' " - по умолчанию "все"), условия (condition=" "- по умолчанию без условий). Вспомогательная функция "add_rec(db, table_name, field_names, field_vals)" предназначена для добавления записей в таблицу.

Основная часть программы начинается с выбора файла базы данных "filename=choose_file(chose_dir='./',file_type='базы данных')". При успешном выборе или создании нового файла заключена в конструкцию контекстного менеджера "with sqlite3.connect(filename[0]) as db:" , позволяющую корректно автоматически завершать работу.

В том случае, если файл базы был открыт как новый в условии "if filename[1]=='create':" создается необходимая структура таблиц с соответствующими полями. Присутствие конструкции "cur.execute("PRAGMA foreign_keys =ON")" обусловлено тем, что по умолчанию в SQLite 3 автоматическая поддержка связанных таблиц отключена.

В основном цикле "while add_continue in ('yes','y','OK','да','продолжить'):" пользователю последовательно предлагается выбрать, добавить новые записи в таблицы или удалить ненужные.

Поскольку данный файл является примером, в нем отсутствуют некоторые вспомогательные функции, имеющиеся в реальных базах данных, такие как детальная проверка вводимых данных, редактирование отдельных полей, проверка связанности таблиц, которая в данном случае осуществляется лишь с использованием встроенных средств. Кроме того, логическая структура интерфейса является упрощенной. Тем не менее на этом примере можно понять принципы основных конструкций, использующиеся при проектировании реальных систем управлении данными.

Разберите подробно код программы и ознакомьтесь с ее работой.

Глава 2. Предварительные математические сведения

2.1. Множества

При описании множеств будем использовать логическую символику, используемую в математике. Так, символом $\forall$ будем заменять выражение "для произвольного", а символом $\exists$ выражение "существует". Эти выражения называются также *кванторами*. Запись $A \Rightarrow B$ (*импликация*) означает, что из справедливости высказывания A вытекает справедливость высказывания B. Если, кроме того, верно и обратное утверждение, то используется символ $A \Leftrightarrow B$. Если предложения A и B справедливы одновременно, то записывают $A \wedge B$. Если же справедливо хотя бы одно из утверждений A или B, то записывают $A \vee B$.

В математике не существует более общего понятия по отношению к понятию «множество», поэтому для него не существует формального определения. Не формально множество можно определить как набор элементов, объединенных по какому-то признаку. Например, множество фруктов может включать в себя яблоки, груши, апельсины. Множество состоит из *элементов*, которые могут быть числами, буквами, словами или другими объектами. Каждый элемент множества *уникален*, то есть в рассматриваемом множестве не может быть повторяющихся элементов. Элементы множества *не имеют* определенного порядка. Множества часто обозначают прописными латинскими буквами, а элементы множеств - строчными. При определении конкретного множества его элементы обозначаются в фигурных скобках либо их перечислением: $A = \{a, b, c, ...\}$, либо указанием их общего свойства: $A = \{x | P(x)\}$ - множество A, состоящее из элементов x, обладающих свойством $P(x)$. Вместо вертикальной черты часто используется двоеточие.

Элемент может *принадлежать* множеству или *не принадлежать*. Для этого используются, соответственно, обозначения $a \in A$ и $a \notin A$. Мощность множества - это количество элементов, которые входят в множество. Множество, которое не содержит ни одного элемента, называется пустым множеством и обозначается символом $\varnothing$.

Если все элементы множества A совпадают с элементами множества B, то такие множества называются равными: $A = B$. Это определение можно также записать формально с использованием логической символики

$$(A=B):=\forall x((x\in A)\Leftrightarrow(x\in B)) \quad . \qquad (1)$$

Множество A является *подмножеством* множества B, если все элементы множества A также являются элементами множества B. Для этого используется обозначение $A\subseteq B$. С использованием логической символики это определение выглядит следующим образом

$$(A\subseteq B):=\forall x((x\in A)\Rightarrow(x\in B)) \quad . \qquad (2)$$

При этом не исключается, что множества A и B могут быть равными. В противном случае множество A называется *собственным* подмножеством множества B, что формально записывается как

$$(A\subset B):=\forall x((A\subseteq B)\wedge(A\neq B)) \quad . \qquad (3)$$

Следует заметить, что в некоторых источниках для обозначения подмножества вместо символа $\subseteq$ используется символ $\subset$. В этом случае для обозначения собственного подмножества рекомендуется использовать символ $\subsetneq$.

В Python для работы с множествами имеется специальный тип переменных set. Для инициализации переменной данного типа используются фигурные скобки, в отличие от прямых, применяемых для обозначения списков (тип переменной list) и круглых для обозначения кортежей (тип tuple).

Ниже приведен фрагмент кода с определением множеств, добавлением в них элементов методом add и удалением из них элементов методом remove.

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8,10,11}
>>> A == B
False
>>> 2 in A
True
>>> 11 in B
True
>>> A.add(6)
>>> A
{1, 2, 3, 4, 5, 6}
>>> B.remove(11)
```

```
>>> 11 in B
False
>>> B
{4, 5, 6, 7, 8, 10}
```

Конструкция x in A проверяет, является ли x элементом множества A. Также имеется метод issubset, проверяющий, является ли данное множество подмножеством другого:

```
>>> A.issubset(B)
False
```

Над множествами определяются *операции*: объединение, пересечение, разность, симметрическая разность. *Объединение* двух множеств $A \cup B$ - это множество, которое содержит все элементы, принадлежащие хотя бы одному из множеств A и B

$$A \cup B = \{x | x \in A \vee x \in B\} \quad . \tag{4}$$

В Python для операции объединения множеств используется метод union

```
>>> C = A.union(B)
>>> C
{1, 2, 3, 4, 5, 6, 7, 8, 10}
```

Обратите внимание, что числа, повторяющиеся в множествах A и B (4,5,6) в множестве C входят только один раз (свойство уникальности элементов). Это свойство на практике удобно применять для выделения из текста не повторяющихся слов. Ниже приведен пример программы такого типа.

```
# Функция для обработки текста
def process_text(text):
    # Удаляем точки и запятые
    cleaned_text = text.replace('.', '').replace(',', '')
     # Преобразуем текст в список слов
    word_list = cleaned_text.split()
    # Преобразуем список в множество для удаления повторяющихся слов
    unique_words = set(word_list)
    return unique_words
```

```
# Пример использования
input_text = "Тридцать три корабля лавировали, лавировали, да не вылавировали."
unique_words_set = process_text(input_text)
# Вывод результата
print(unique_words_set)
{'вылавировали', 'да', 'не', 'корабля', 'три', 'Тридцать', 'лавировали'}
```

Пересечение двух множеств $A \cap B$ - это множество, которое содержит только те элементы, которые принадлежат обоим множествам

$$A \cap B = \{x | x \in A \wedge x \in B\} \quad . \tag{5}$$

В Python для операции пересечения множеств используется метод intersection

```
>>> D = A.intersection(B)
>>> D
{4, 5, 6}
```

Как видно, в множество D вошли только элементы, повторяющиеся в множествах A и B. Для проверки того, что два множества не пересекаются, используется метод isdisjoint:

```
>>> A.isdisjoint(B)
False
```

Объединение и пересечение *n* множеств обозначается, соответственно, как $\bigcup A_n$ и $\bigcap A_n$.

Разность двух множеств $A \setminus B$ - это множество, которое содержит все элементы первого множества, которые не принадлежат второму множеству

$$A \setminus B = \{x | x \in A \wedge x \notin B\} \quad . \tag{6}$$

В Python для операции разности множеств используется метод difference

```
>>> E = A.difference(B)
>>> E
{1, 2, 3}
```

Симметрическая разность двух множеств $A \Delta B$ - это множество, которое содержит все элементы обоих множеств, кроме их пересечения

$$A \Delta B = (A \setminus B) \cup (B \setminus A) \quad . \tag{7}$$

Пример вычисления симметрической разности в Python:

```
>>> A_sd_B = A.difference(B).union(B.difference(A))
>>> A_sd_B
{1, 2, 3, 7, 8, 10}
```

Дополнение множества A до некоторого более общего множества S обозначается CA или $\bar{A}$

$$\bar{A} = S \setminus A \quad . \tag{8}$$

Пусть имеется некоторое множество Ω. Система $\mathcal{F}$ подмножеств множества Ω называется *алгеброй*, если $\Omega \in \mathcal{F}$ и объединение, пересечение и разность любых множеств системы $\mathcal{F}$ также принадлежат этой системе. Формально это определение можно записать следующим образом

$$\begin{array}{l} \text{a) } \Omega \in \mathcal{F} \\ \text{b) } A, B \in \mathcal{F} \Rightarrow A \cup B \in \mathcal{F}, A \cap B \in \mathcal{F} \\ \text{c) } A, B \in \mathcal{F} \Rightarrow A \setminus B \in \mathcal{F}, B \setminus A \in \mathcal{F} \quad . \end{array} \tag{9}$$

Если в определении (9) расширить свойство b) до *n* подмножеств

$$\text{b) } A_n \in \mathcal{F} \Rightarrow \bigcup A_n \in \mathcal{F}, \bigcap A_n \in \mathcal{F} \quad , \tag{10}$$

то такая система называется *σ-алгеброй*.

Пусть на множестве Ω задана некоторая σ-алгебра его подмножеств $\mathcal{F}$. Числовая функция $\mu(A)$, определенная на множествах $A \subset \mathcal{F}$, называется *мерой, заданной на* $\mathcal{F}$, если

1. $\forall A \subset \mathcal{F}$ выполнено $\mu(A) \geq 0$;

2. для любого счетного набора попарно непересекающихся множеств $A_n \subset \mathcal{F}$, $n = 1,2,\ldots$, таких, что $\bigcup_{n=1}^{\infty} A_n \subset \mathcal{F}$, выполнено свойство счетной аддитивности (σ-аддитивности): $\mu\left(\bigcup_{n=1}^{\infty} A_n \right) = \sum_{n=1}^{\infty} \mu(A_n)$.

Мера μ называется конечной, если дополнительно выполнено условие $\mu(A)<\infty$.

Мера μ , заданная на σ-алгебре $\mathcal{F}$, обладает следующими свойствами:

1. $\mu(\varnothing)=0$;
2. $\mu(A)\le\mu(B)$ для $A,B\subset\mathcal{F}$, таких, что $A\subseteq B$;
3. $\mu(A\cup B)=\mu(A)+\mu(B)-\mu(A\cap B)$ для всех $A,B\subset\mathcal{F}$;
4. Если $A_n\subset\mathcal{F}$, $n=1,2,\ldots$ - убывающая последовательность множеств, т.е. $A_1\supseteq A_2\supseteq\ldots$, такая, что $\mu(A_1)<\infty$ и $\bigcap\limits_{n=1}^{\infty}A_n\subset\mathcal{F}$, то $\mu\left(\bigcap\limits_{n=1}^{\infty}A_n\right)=\lim\limits_{n\to\infty}\mu(A_n)$;
5. Если $A_n\subset\mathcal{F}$, $n=1,2,\ldots$ - возрастающая последовательность множеств, т.е. $A_1\subseteq A_2\subseteq\ldots$, такая, что $\bigcup\limits_{n=1}^{\infty}A_n\subset\mathcal{F}$, то $\mu\left(\bigcup\limits_{n=1}^{\infty}A_n\right)=\lim\limits_{n\to\infty}\mu(A_n)$.

Множество X вместе с мерой μ , определенной на некоторой σ-алгебре $\mathcal{F}$, называется *пространством с мерой* и обозначается $(\Omega,\mathcal{F},\mu)$.

Всякую σ-алгебру $\mathcal{F}$ можно пополнить множествами вида $A\cup N$, где $A\subset\mathcal{F}$, а $N\subset A_0$ для некоторого $A_0\subset A$, имеющего нулевую меру: $\mu(A_0)=0$. Получаемое в результате пространство с мерой называется *полным*.

2.2. Аксиомы теории вероятностей

Теория вероятностей начала формироваться в современном понимании в середине XVII века, когда математики, такие как Блез Паскаль и Пьер де Ферма, начали исследовать проблемы, связанные с азартными играми. Эти исследования положили начало более систематическому изучению вероятностей.

Однако аксиоматическое определение теории вероятностей, которое мы знаем сегодня, было предложено академиком Андреем Николаевичем Колмогоровым в 1930-х годах. Его работа в этом

направлении значительно повлияла на дальнейшее развитие теории и сделала её более строгой и математически обоснованной. Систематическое изложение теории вероятностей было дано А.Н. Колмогоровым в книге [22] и затем вошло в современные учебники [1, 23].

Приведем аксиомы Колмогорова, используя определения (9),(10)

Пусть имеется множество Ω элементов ω, называемых *элементарными событиями*, и σ-алгебра $\mathcal{F}$ подмножеств множества Ω. Элементы множества $\mathcal{F}$ называются *случайными событиями* или просто *событиями*.

A1. $\mathcal{F}$ является *алгеброй событий*.

A2. Каждому событию A из $\mathcal{F}$ поставлено в соответствие неотрицательное вещественное число $P(A)$, называемое *вероятностью* события A (в современном изложении $P(A)$ - нормированная мера на $\mathcal{F}$).

A3. $P(\Omega)=1$.

A4. Если A и B не пересекаются, то $P(A+B)=P(A)+P(B)$

Совокупность объектов $(\Omega,\mathcal{F},P)$ называется *вероятностным пространством*. Из приведенных аксиом вытекает, что определение вероятности в существенной мере опирается на теорию множеств. В таблице 1 приведены основные соответствия [1] между понятиями теории вероятностей и теории множеств.

Перечислим свойства вероятности P, которые вытекают из ее определения и свойств меры.

1. $0\le P(A)\le 1$ для любого события $A\in\mathcal{F}$;
2. $P(\varnothing)=0$, где $\varnothing=\overline{\Omega}$ - невозможное событие;
3. $P(A+B)=P(A)+P(B)-P(AB)$ для любых событий $A,B\in\mathcal{F}$, где $AB=A\cap B$
4. $P(A)\le P(B)$, если $A\subseteq B$ (т.е. A - частный случай события B).

Таблица 1. Соответствие понятий теории множеств и теории вероятностей

Обозначения	Интерпретация в теории множеств	Интерпретация в теории вероятностей
ω	элемент, точка	исход, элементарное событие
Ω	множество точек	пространство исходов, элементарных событий; достоверное событие
$\mathcal{F}$	σ-алгебра $\mathscr{F}$ подмножеств	σ-алгебра $\mathscr{F}$ событий
$A \in \mathcal{F}$	множество точек	событие (если исход $\omega \in A$, то говорят, что наступило событие А)
$\bar{A} = \Omega \setminus A$	дополнение множества А , т. е. множество точек ω, не входящих в А	событие, состоящее в ненаступлении события А
$A \cup B$ или $A + B$	объединение множеств А и В, т.е. множество точек ω, входящих или в А, или в В	событие, состоящее в том, что произошло по крайней мере одно из событий А или В
$A \cap B$ или AB	пересечение множеств А и В, т. е. множество точек ω, входящих и в А , и в В	событие, состоящее в том, что одновременно произошло и А , и В
$\varnothing$	пустое множество	невозможное событие
$A \cap B = \varnothing$	множества А и В не пересекаются	события А и В несовместны (не могут наступать одновременно)
$A \setminus B$	разность множеств А и В, т. е. множество точек, входящих в А , но не входящих в В	событие, состоящее в том, что произошло А , но не произошло В
$A \Delta B$	симметрическая разность множеств	событие, состоящее в том, что произошло одно из событий А или В , но не оба одновременно
$\sum_n A_n$	сумма, т. е. объединение попарно непересекающихся множеств	событие, состоящее в наступлении одного из несовместных событий $A_1, A_2, \ldots$

Пусть A и B - случайные события, причем $P(B)>0$. Тогда отношение $P(AB)/P(B)$ называется условной вероятностью события A при условии B и обозначается $P(A|B)$. Таким образом,

$$P(A|B)=\frac{P(AB)}{P(B)} \quad . \tag{11}$$

Аксиоматическое определение вероятности позволяет использовать для решения задач теории вероятностей современный математический аппарат теории функций и функционального анализа. При таком подходе при определении закона распределения случайной величины и его параметров применяется интеграл Лебега. Это дает возможность одним и тем же способом описывать как дискретные, так и непрерывные случайные величины. Мы, однако, в дальнейшем будем иметь дело только с непрерывными распределениями случайных величин, так что для их описания достаточно будет использования обычного интеграла Римана.

С понятием случайного события непосредственно связано понятие *случайной величины*, которая служит для количественного описания результатов случайных событий. Случайная величина — это переменная, значения которой представляют собой численные исходы некоторого случайного феномена или эксперимента. В дальнейшем случайные величины будем обозначать заглавными латинскими буквами, а их значения, которые они принимают в конкретном событии - соответствующими им строчными буквами. Например, x - конкретное значение случайной величины X . В отличие от неслучайных, детерминированных величин, для случайной величины нельзя заранее точно сказать, какое конкретное значение она примет в определенных условиях, а можно только указать закон ее распределения. Закон распределения считается заданным, если: 1) указано множество возможных значений случайной величины; 2) указан способ количественного определения вероятности попадания случайной величины в любую область множества возможных значений.

Если множество возможных значений случайной величины имеет дискретный характер, так что каждое возможное значение отделено от соседних некоторым числовым промежутком, то такая случайная величина называется *дискретной*. Например, множество возможных значений

дискретной случайной величины - числа выпавших очков на игральной кости составляет {1, 2, 3, 4, 5, 6}. Термин *дискретная* показывает, что случайная величина может принять лишь конечное или счетное множество значений. Случайные величины, множество значений которых составляет некоторый непрерывный конечный или бесконечный промежуток (область), называются *непрерывными*. Например, непрерывной случайной величиной является температура в производственном химическом реакторе. Множество возможных значений непрерывной случайной величины бесконечно и имеет мощность континуума. Непрерывные случайные величины наиболее часто встречаются в технике.

Помимо аксиоматического существует также *классическое определение вероятности* случайной величины, которое служило основой теории вероятностей со времен ее зарождения в XVII столетии (Паскаль, Ферма, Бернулли) до того как А.Н. Колмогоровым был сформулирован аксиоматический подход к ее определению. В рамках классического определения рассматриваются дискретные случайные величины. Каждое значение дискретной случайной величины связывается с определенным случайным событием. Под вероятностью $P(A)$ случайного события A понимается отношение числа *m* элементарных исходов, благоприятствующих событию A, к общему числу *n* всех равновозможных несовместных элементарных исходов, образующих полную группу

$$P(A)=\frac{m}{n} \quad . \tag{12}$$

2.3. Функции распределения вероятностей

Ключевым понятием в теории вероятностей и математической статистике является *распределение вероятностей*, которое описывает, как распределены вероятности различных исходов случайной величины. Оно определяет область значений случайной величины и соответствующие вероятности для каждого из этих значений. Существуют два основных типа распределений вероятностей. *Дискретные* распределения вероятностей применяются к дискретным случайным величинам. Например, количество очков при бросании кубика или число угаданных номеров в лотерее. Каждое значение имеет определенную вероятность, и сумма всех вероятностей равна единице (см. **А3**). *Непрерывные* распределения вероятностей относятся к

непрерывным случайным величинам. В отличие от дискретных, для непрерывных распределений вероятность того, что случайная величина примет конкретное значение, равна нулю.

Количественное описание распределений вероятности осуществляется с помощью *функций распределения*.

2.3.1. Одномерные случайные величины

В рамках аксиоматического определения вероятности можно сформулировать определение непрерывной случайной величины. Пусть $(\Omega, \mathcal{F}, P)$ - вероятностное пространство. Будем рассматривать числовые функции $X = X(\omega)$, заданные при всех $\omega \in \Omega$ или на множестве вероятности 1. Числовая функция $X = X(\omega)$ называется *случайной величиной*, если для любого x, $-\infty < x < \infty$, множество тех ω, для которых $X(\omega) < x$, принадлежит σ-алгебре $\mathcal{F}$ и $P(-\infty < X(\omega) < \infty) = 1$. Аналитическими выражениями законов распределения случайных величин являются функции распределения вероятностей — интегральная и дифференциальная.

Интегральная функция распределения $F(x)$ случайной величины X показывает вероятность того, что случайная величина не превышает некоторого заданного или текущего значения x т. е. $F(x) = P(X \leqslant x)$. Следовательно, вероятность того, что значение случайной величины X заключено между x_1 и x_2, равна разности значений функции распределения, вычисленных в этих двух точках:

$$P(x_1 < X \leq x_2) = F(x_2) - F(x_1) \quad . \tag{13}$$

Аналогично,

$$P(X > x) = 1 - F(x) \quad . \tag{14}$$

Интегральная функция распределения случайной величины X обладает следующими свойствами:

1) $\lim\limits_{x\to-\infty} F(x)=F(-\infty)=0$ 2) $\lim\limits_{x\to\infty} F(x)=F(\infty)=1$

3) $F(x)\geq 0$ для всех x 4) $F(x_2)\geq F(x_1)$, если $x_2>x_1$.

Если функция $F(x)$ дифференцируема для всех значений случайной величины *X*, то закон распределения вероятностей может быть выражен в аналитической форме также с помощью *дифференциальной функции* распределения вероятностей

$$f(x)=\frac{dF(x)}{dx}=\lim_{\Delta x\to 0}\frac{P(x<X\leq x+\Delta x)}{\Delta x} \quad . \qquad (15)$$

Таким образом, значение функции $f(x)$ приближенно равно отношению вероятности попадания случайной величины в интервал $(x, x+\Delta x)$ к длине Δx этого интервала, когда Δx — бесконечно малая величина. Поэтому функцию $f(x)$ называют также *функцией плотности распределения вероятностей* (или *короче — функцией плотности вероятности).*

Основные свойства функции $f(x)$ *:*

1) $f(x)\geq 0$; 2) $\int\limits_{-\infty}^{\infty} f(x)dx=1$;

2) $\int\limits_{-\infty}^{x} f(z)dz=F(x)$; 3) $\lim\limits_{|x|\to\infty} f(x)=0$.

С помощью дифференциальной функции распределения вычисляется вероятность нахождения случайной величины в любой области из множества ее возможных значений. В частности,

$$P(X \le a_1) = \int_{-\infty}^{a_1} f(x)\,dx \quad ,$$

$$P(X > a_2) = \int_{a_2}^{\infty} f(x)\,dx \quad , \qquad (16)$$

$$P(a_1 < X \le a_2) = \int_{a_1}^{a_2} f(x)\,dx \quad .$$

Для непрерывной случайной величины вероятность можно определить как относительную долю площади под кривой плотности распределения вероятностей $f(x)$. Так, например, вероятность того, что случайная величина X примет значение, меньшее a_1 , равна относительной доле площади под кривой $f(x)$ слева от точки a_1 , вероятность того, что эта величина *X* примет значение, большее a_2 , равна относительной доле площади под кривой $f(x)$ справа от точки a_2 ; вероятность того, что она примет значение, заключенное между a_1 и a_2 , равна относительной доле площади под кривой $f(x)$ между точками a_1 и a_2.

Как интегральная, так и дифференциальная функции распределения являются исчерпывающими вероятностными характеристиками случайной величины. Однако некоторые основные свойства случайных величин могут быть описаны более просто с помощью определенных числовых параметров. Наибольшую роль среди них на практике играют два параметра, характеризующие центр рассеяния (центр распределения) случайной величины и степень ее рассеяния вокруг этого центра. Наиболее распространенной характеристикой центра распределения является *математическое ожидание* m_x случайной величины X (часто называемое также *генеральным средним значением*) :

$$m_x = \int_{-\infty}^{\infty} x f(x)\,dx \quad . \qquad (17)$$

Для обозначения математического ожидания случайной величины X также используется обозначение EX .

Степень рассеяния случайной величины X относительно m_x, может быть охарактеризована с помощью *генеральной дисперсии* σ_x^2 :

$$\sigma_x^2 = \int_{-\infty}^{\infty} (x - m_x)^2 f(x)\, dx \quad . \tag{18}$$

Если $f(x)$ все в большей степени концентрируется вблизи m_x, то значения σ_x^2 уменьшаются. Если же имеются весьма удаленные от m_x значения случайной величины X и для них $f(x)$ не слишком мала, то дисперсия σ_x^2 увеличивается. Квадратный корень из дисперсии называется *средним квадратическим отклонением* и обозначается σ_x .

Сравнивая формулы (17) и (18), можно сказать, что дисперсия представляет собой математическое ожидание квадрата отклонения случайной величины от ее математического ожидания. По аналогии можно ввести параметры более высокого порядка. Они называются центральными моментами. Центральный момент порядка r случайной величины вычисляется по формуле

$$\mu_r = \int_{-\infty}^{\infty} (x - m_x)^r f(x)\, dx \quad . \tag{19}$$

Кроме дисперсии, являющейся центральным моментом второго порядка, на практике используются центральные моменты третьего и четвертого порядка.

Центральный момент третьего порядка характеризует несимметричность распределения относительно математического ожидания. Поэтому за характеристику несимметричности распределения принимают безразмерную величину - отношение центрального момента третьего порядка к кубу среднеквадратического отклонения, называемую *асимметрией* распределения:

$$As = \frac{\mu_3}{\sigma_x^3} = \frac{\mu_3}{\mu_2^{\frac{3}{2}}} \tag{20}$$

Центральный момент четвертого порядка определяет характер кривой распределения в точке максимума: "островершинность" или "плосковершинность" распределения. Безразмерная величина, вычисляемая на его основе по формуле

$$Ek = \frac{\mu_4}{\sigma_x^4} - 3 = \frac{\mu_4}{\mu_2^2} - 3 \qquad (21)$$

называется *эксцессом* распределения.

Рассмотрим несколько примеров одномерных распределений случайных величин.

Равномерное распределение. Пусть вероятность попадания случайной величины X в каждый бесконечно малый интервал некоторого отрезка $x \in [a, b]$ постоянна, т.е. равна некоторой константе C, а вероятность того, что она примет любое значение, не принадлежащее этому отрезку, равна нулю. Плотность распределения такой случайной величины можно определить следующим образом:

$$f(x) = \begin{cases} C, при\ x \in [a, b], \\ 0, при\ x \geq 0 \notin [a, b] \end{cases} . \qquad (22)$$

Пользуясь свойством 2) дифференциальной функции распределения (условием нормировки плотности распределения), определим значение константы C:

$$\int_{-\infty}^{\infty} f(x)\,dx = 1 \quad \Rightarrow \quad \int_{a}^{b} C\,dx = 1 \quad \Rightarrow \quad C = \frac{1}{b-a} .$$

Подставив вычисленное значение константы в выражение (22), получим в явном виде вид дифференциальной функции равномерного распределения

$$f(x) = \begin{cases} \frac{1}{b-a}, при\ x \in [a, b], \\ 0, при\ x \geq 0 \notin [a, b] \end{cases} . \qquad (23)$$

Пользуясь свойством 3) дифференциальной функции распределения (связь между дифференциальной и интегральной функциями), получим в явном виде интегральную функцию равномерного распределения

$$F(x)=\int_{-\infty}^{x} f(z)\,dz = \begin{cases} 0, & при\ \ x \leqslant a \\ \dfrac{x-a}{b-a}, & при\ \ a<x<b, \\ 1, & при\ \ x \geq b \end{cases} . \qquad (24)$$

Функцию $F(x)$ можно использовать для того, чтобы определить вероятность попадания случайной величины X в некоторый конечный интервал $[x_1, x_2]$, лежащий внутри интервала $[a, b]$. Подставляя (24) в формулу (13), найдем, что эта вероятность будет равна

$$P(x_1 \leq X \leq x_2)=\frac{x_2-x_1}{b-a} .$$

Заметьте, что если интервал $[x_1, x_2]$ не пересекается с интервалом $[a, b]$, т.е. $x_2<a$ или $x_1>b$, то для вероятности попадания случайной величины в такой интервал формулы (24) и (13) дают значение 0.

Вычислим математическое ожидание и дисперсию равномерного распределения. Подставив выражение функции распределения (23) в формулу математического ожидания (17) и выполнив интегрирование, получим

$$m_x=\frac{b-a}{2} . \qquad (25)$$

Выполнив аналогичные вычисления для дисперсии, используя формулу (18), будем иметь

$$\sigma_x^2=\frac{a^2-2ab+b^2}{12} . \qquad (26)$$

Ниже приведен вариант кода программы для построения графиков дифференциальной и интегральной функций равномерного распределения.

```
import numpy as np
```

```
import matplotlib.pyplot as plt

def f(x,a,b):
    if x>=a and x<=b:
        return 1/(b-a)
    else:
        return 0
def F(x,a,b):
    if x<a:
        return 0
    elif x>=a and x<=b:
        return (x-a)/(b-a)
    else:
        return 1
x=np.linspace(0,4,200)
a=1
b=3
y_f=[]
y_F=[]
for i in range(np.size(x)):
    y_f.append(f(x[i],a,b))
    y_F.append(F(x[i],a,b))
plt.plot(x,y_f,x,y_F)
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend(['f(x)', 'F(x)'])
plt.show()
```

В данной программе определяются две функции: f(x,a,b), F(x,a,b) для вычисления, соответственно, дифференциальной и интегральной функций распределения при заданных параметрах a, b. Массив из 200 значений x в интервале от 0 до 4 вычисляется с помощью функции linspace библиотеки numpy. После ввода значений параметров, создаются списки y_f, y_F со

значениями функций в точках массива x и строятся графики с использованием библиотеки matplotlib.pyplot. На рисунке приведен пример графика.

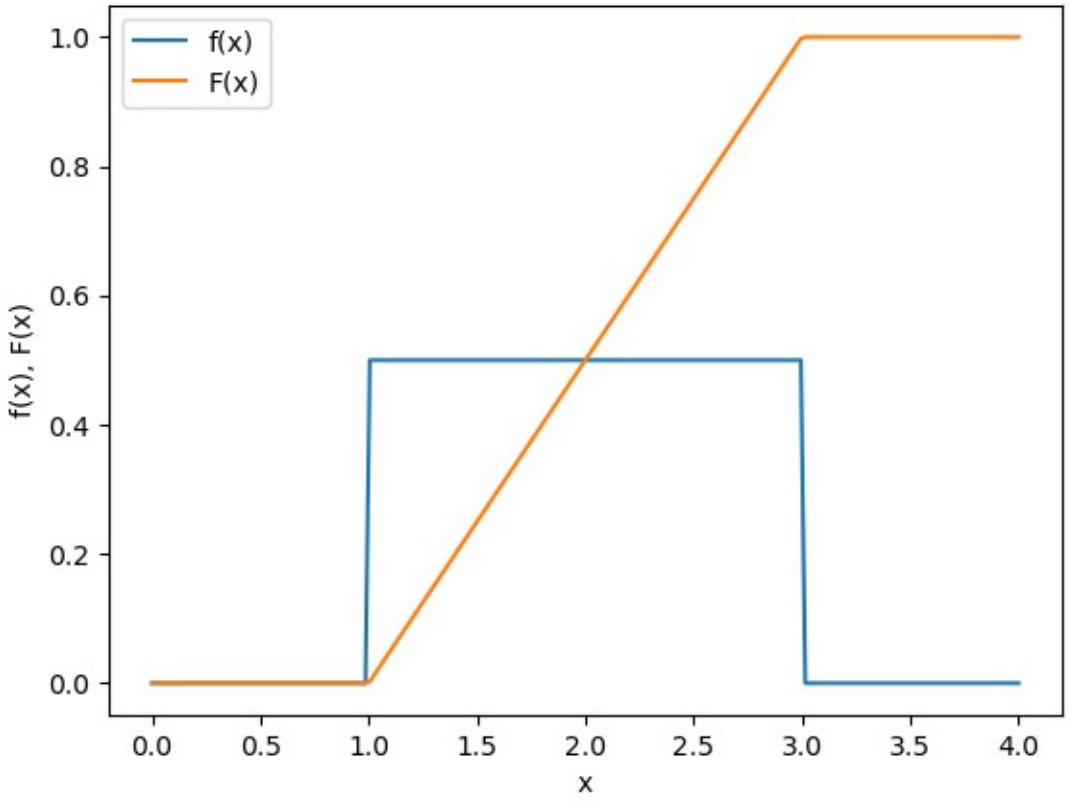

Рисунок 1. Пример графиков функций равномерного распределения

В практике математического моделирования часто возникают задачи, в которых требуется генерирование случайных чисел с определенным распределением. В большинстве универсальных языков программирования, а также программных пакетах автоматизации вычислений, имеются функции для генерирования так называемых псевдослучайных чисел. Псевдослучайные числа — это последовательность чисел, которая вычисляется по определённому арифметическому правилу, но имеет все свойства случайной последовательности чисел в рамках решаемой задачи. Хотя псевдослучайные числа могут казаться случайными, они не являются истинно случайными. В реальности, любой псевдослучайный генератор с конечным числом внутренних состояний повторится после очень длинной последовательности чисел.

В Python имеется несколько возможностей для генерации псевдослучайных чисел. В простейшем варианте можно использовать встроенный модуль random, имеющий несколько

функций для генерации псевдослучайных вещественных и целых чисел, а также для работы со списками с применением случайных чисел. Функция random() модуля random использует алгоритм Мерсенна для генерации псевдослучайных чисел, равномерно распределенных в интервале от 0 до 1. Для инициализации генератора псевдослучайных чисел используется функция random.seed([a]), где параметр a используется для старта псевдослучайной последовательности. Если он не задан, для инициализации последовательности используется текущее системное время. Ниже приведены примеры вызова функций для генерации случайных чисел с использованием данного модуля.

```
>>> import random
>>> random.seed(111)
# Генерация псевдослучайного числа в диапазоне от 0 до 1
>>> random.random()
0.21276311517617263
# Генерация псевдослучайного целого числа в диапазоне от 0 до 10
>>> random.randint(0, 10)
5
```

Для более эффективной работы со случайными числами целесообразно использовать модуль random библиотеки numpy, предлагающий более широкий спектр функций для работы с массивами случайных чисел.

Нормальное распределение (распределение Гаусса). Это распределение является важнейшим в теории вероятностей. Оно описывается плотностью

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-m}{\sigma}\right)^2} \quad . \tag{27}$$

Интегральная функция нормального распределения имеет вид

$$F(x)=\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{1}{2}\left(\frac{z-m}{\sigma}\right)^2} dz \quad . \tag{28}$$

Мы видим, что нормальное распределение определяется двумя параметрами: m и σ. Подставляя плотность распределения в формулы (12) и (13), несложно убедиться, что эти

параметры представляют собой соответственно математическое ожидание и среднеквадратическое отклонение данного распределения.

Широкое распространение на практике нормального распределения обусловлено тем обстоятельством, что им во многих случаях описывается распределение ошибок наблюдений. Поэтому нормальное распределение называют также распределением ошибок. Объяснение этого факта дает центральная предельная теорема теории вероятностей: *если случайная величина представляет собой сумму очень большого числа независимых случайных величин, влияние каждой из которых на всю сумму ничтожно мало, то имеет распределение, близкое к нормальному*.

На практике часто используется нормированное нормальное распределение, которое получается из формулы (28) заменой переменных $x = m + \sigma z$, $z = (x - m)/\sigma$. Интегральная функция нормированного нормального распределения имеет вид

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-z^2/2} dz \quad . \tag{29}$$

Для вычисления интегральной и дифференциальной функций нормального распределения в Python можно использовать библиотеку «scipy.stats», которая предоставляет удобные методы для работы с различными статистическими распределениями, включая нормальное распределение. Ниже приведены фрагменты кода для иллюстрации функций данной библиотеки. В качестве примера вычисляются функции нормированного нормального распределения.

```
# Импорт необходимых библиотек
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
# Параметры нормального распределения: среднее и стандартное отклонение
mu = 0      # Среднее
sigma = 1   # Стандартное отклонение
# Вычисление дифференциальной функции (PDF)
x = np.linspace(-5, 5, 100)
pdf = stats.norm.pdf(x, mu, sigma)
plt.plot(x, pdf, label='f(x)')
```

```
# Вычисление интегральной функции (CDF)
#Интегральная функция распределения (CDF) может быть вычислена с помощью метода «cdf»:
cdf = stats.norm.cdf(x, mu, sigma)
plt.plot(x, cdf, label='F(x)')
plt.title('Функции нормального распределения')
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend()
plt.grid()
plt.show()
# Пример использования функции «cdf» для нахождения вероятности того, что случайная величина X будет меньше или равна некоторому значению x_0:
x0 = 1.5
probability = stats.norm.cdf(x0, mu, sigma)
print(f'Вероятность того, что X ≤ {x0}: {probability:.4f}')
```

Результат:

Вероятность того, что X ≤ 1.5: 0.9332

Графики функций приведены на рисунке 2. Для симметричного распределения третий момент и, соответственно, коэффициент асимметрии распределения равны нулю. Отношение четвертого момента к квадрату дисперсии нормального распределения, как можно убедиться прямым вычислением, равно трем. Поэтому коэффициент эксцесса, вычисляемый по формуле (21), для нормального распределения равен нулю. Заметим, что слагаемое -3 входит в формулу для эксцесса именно для того, чтобы при сравнении кривых различных распределений за ноль принять эксцесс нормального распределения.

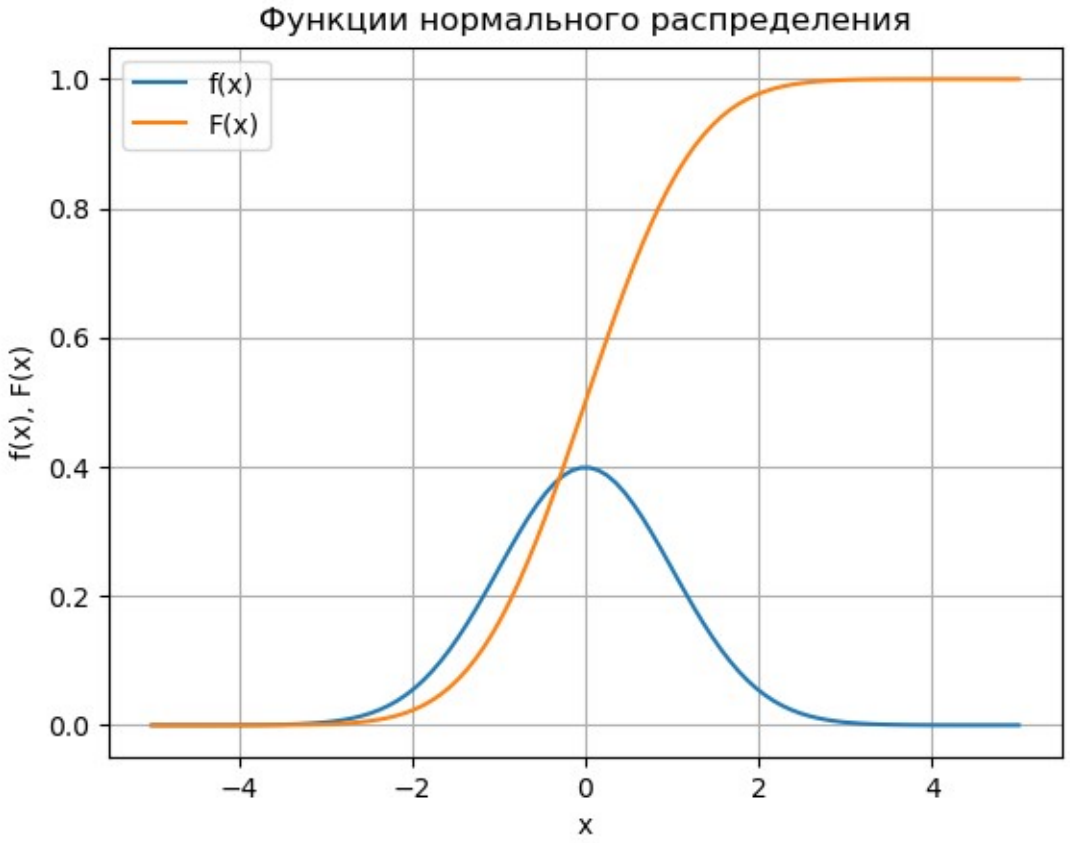

Рисунок 2. Графики функций нормированного нормального распределения

В практических задачах часто возникает необходимость по известному значению интегральной функции распределения найти отвечающее ему значение аргумента, т.е. решить уравнение

$$F(x)=p \quad , \tag{30}$$

где p - заданная величина .

Корень x_p этого уравнения называется *квантилем* распределения порядка . Он соответствует значению x_p случайной величины X , при котором ее вероятность не превысит заданного значения p :

$$P\left\{X < x_p\right\} = p \quad . \tag{31}$$

Квантиль $x_{\frac{1}{2}}$ называется медианой распределения. Ордината медианы рассекает площадь между кривой плотности вероятности и осью абсцисс пополам. Если распределение симметрично, то $x_{\frac{1}{2}} = m_x$.

Для вычисления квантиля нормального распределения в Python можно использовать функцию «scipy.stats.norm.ppf()» (Percent Point Function), которая является обратной к функции распределения (CDF).

Функция ppf() принимает вероятность (p) в качестве аргумента и возвращает значение квантиля (x) для заданного нормального распределения с параметрами «mu» и «sigma»:

```
from scipy.stats import norm
p = 0.95  # Вероятность
mu = 0    # Среднее
sigma = 1 # Стандартное отклонение
x = norm.ppf(p, loc=mu, scale=sigma)
print(f'Квантиль уровня {p}: {x:.4f}')
```

Этот код вычисляет 95-й процентиль (квантиль уровня 0.95) стандартного нормального распределения (mu=0, sigma=1), который приблизительно равен 1.6449. Для нормального распределения с произвольными параметрами «mu» и «sigma» можно использовать параметры функции norm.ppf «loc» и «scale» соответственно.

Модуль random библиотеки NumPy предоставляет функции «numpy.random.normal()» для генерации случайных чисел, подчиняющихся нормальному распределению, и «numpy.random.Generator.normal()» для генерации массивов случайных чисел с нормальным распределением.

Распределения вероятностей отказов. В теории надежности применяется понятие вероятности отказа некоторого элемента или устройства. Пусть имеется элемент со случайным временем X безотказной работы, причем существует непрерывная плотность распределения

$f(x)$. Вычислим вероятность $P(x,\Delta x)$ того, что элемент откажет в интервале времени $(x,\ x+\Delta x)$ при условии, что он проработал без отказа до момента x . Пользуясь формулой условной вероятности (11) и формулами (13) и (14), получим

$$P(x,\Delta x)=P(x<X<x+\Delta x \mid X\geq x)=\frac{F(x+\Delta x)-F(x)}{1-F(x)} \quad .$$

Найдем отношение этой вероятности к интервалу Δx :

$$\frac{P(x,\Delta x)}{\Delta x}=\frac{1}{1-F(x)}\,\frac{F(x+\Delta x)-F(x)}{\Delta x} \quad .$$

При стремлении Δx к нулю будем иметь

$$\lim_{\Delta x\to 0}\frac{P(x,\Delta x)}{\Delta x}=\frac{1}{1-F(x)}\lim_{\Delta x\to 0}\frac{F(x+\Delta x)-F(x)}{\Delta x}=\frac{f(x)}{1-F(x)} \quad .$$

Величина

$$\lambda(x)=\frac{f(x)}{1-F(x)} \tag{32}$$

называется *интенсивностью* случайной величины X . В теории надежности она называется *интенсивностью отказа* элемента, поскольку $\lambda(x)dx$ есть вероятность отказа элемента за бесконечно малый промежуток времени dx при условии безотказной работы элемента в течение времени x .

Равенство (32) можно записать в виде дифференциального уравнения

$$\frac{F'(x)}{1-F(x)}=\lambda(x) \quad . \tag{33}$$

Интегрируем это уравнение, считая $t>0$,

$$\int_0^t \frac{F'(x)dx}{1-F(x)} = -\ln(1-F(x))\Big|_0^t = -\ln(1-F(t)) = \int_0^t \lambda(x)dx \quad .$$

Отсюда

$$F(t) = 1 - exp\left\{-\int_0^t \lambda(x)dx\right\} \quad , \quad x>0 \quad . \tag{34}$$

Таким образом, мы получили выражение для функции вероятности отказа $F(t)$. Пусть интенсивность отказов не зависит от времени $\lambda(x) = const = \lambda$ при всех $x>0$. Тогда из уравнения (34) находим

$$F(t) = 1 - e^{-\lambda t} \quad . \tag{35}$$

Распределение случайной величины X , для которого

$$F(x) = \begin{cases} 1-e^{-\lambda x}, & x>0 \\ 0, & x \leqslant 0 \end{cases} \quad ,$$

называется *экспоненциальным* (*показательным*). Константа λ называется параметром экспоненциального распределения. Плотность экспоненциального распределения имеет вид экспоненты

$$f(x) = F'(x) = \lambda e^{-\lambda x} \quad . \tag{36}$$

Экспоненциальное распределение встречается не только в теории надежности. Им описываются многие случайные процессы в природе, например, вероятность радиоактивного распада, вероятность гибели бактерий и т. д.

Вычислим математическое ожидание и дисперсию экспоненциального распределения. Подставляя функцию (36) в выражение (17), получим

$$m_x = \int_{-\infty}^{\infty} x f(x)\,dx = \int_{0}^{\infty} x\lambda e^{-\lambda x}\,dx = \frac{1}{\lambda} \quad .$$

Для дисперсии с помощью формулы (18) и вычисленного математического ожидания получим

$$\sigma_x^2 = \int_{0}^{\infty} \left(x - \frac{1}{\lambda}\right)^2 \lambda e^{-\lambda x}\,dx = \frac{1}{\lambda^2} \quad .$$

Используя формулы (20) и (21), можно убедиться, что асимметрия и эксцесс экспоненциального распределения не зависят от параметра λ и имеют значения соответственно $As = 2$, $Ek = 6$.

Если в качестве $\lambda(x)$ в уравнении (34) взять степенную функцию, то для $F(x)$ придем к распределению *Вейбулла*

$$F(x) = \begin{cases} 1 - e^{-\left(\frac{x}{b}\right)^a}, & x > 0 \\ 0, & x \leqslant 0 \end{cases} , \qquad (37)$$

где параметры распределения: a - коэффициент формы, b - коэффициент масштаба.

Плотность распределения Вейбулла имеет вид

$$f(x) = F'(x) = \begin{cases} a\,b^{-a}\,x^{a-1} e^{-\left(\frac{x}{b}\right)^a}, & x > 0 \\ 0, & x \leqslant 0 \end{cases} . \qquad (38)$$

Распределение Вейбулла широко применяется в теории надежности. Оно универсально: при $a = 1$ распределение Вейбулла переходит в экспоненциальное, а при $a = 2$ похоже на нормальное.

Для значения $a = 2$ несложно вычислить моменты распределения. Так, для математического ожидания имеем

$$m_x = \int_{-\infty}^{\infty} x f(x)\,dx = \frac{2}{b^2}\int_0^{\infty} x^2 e^{-(x/b)^2}\,dx = \frac{b}{2}\sqrt{\pi} \quad . \tag{39}$$

Формула (39) дает выражение наиболее вероятного времени безотказной работы изделия.

Аналогичным образом с помощью формулы (18) вычислим дисперсию распределения Вейбулла:

$$\sigma_x^2 = \int_{-\infty}^{\infty} (x - m_x)^2 f(x)\,dx = \frac{2}{b^2}\int_0^{\infty} \left(x - \frac{b}{2}\sqrt{\pi}\right)^2 x e^{-(x/b)^2}\,dx = b^2\left(1 - \frac{\pi}{4}\right) \quad . \tag{40}$$

Как следует из полученных результатов, при $a = 2$ с увеличением параметра b время безотказной работы изделия будет увеличиваться, и при этом будет увеличиваться разброс значений долговечности.

В библиотеке scipy имеются функции weibull_min.pdf(x, a, scale=b) и weibull_min.cdf(x, a, scale=b) для вычисления, соответственно, дифференциальной и интегральной функций распределения Вейбулла. Ниже приведен вариант кода для построения графиков этих функций.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import weibull_min
def f(x,a,b):
    return weibull_min.pdf(x, a, scale=b)
def F(x,a,b):
    return weibull_min.cdf(x, a, scale=b)
a=1.5 # коэффициент формы
b= 1  # коэффициент масштаба
x = np.linspace(0, 5, 100)
plt.plot(x,f(x,a,b),x,F(x,a,b))
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend(['f(x)', 'F(x)'])
```

```
plt.title(f'Функции распределения Вейбулла: a={a}, b={b}')
plt.show()
```

Примеры графиков для различных значений параметра «a» привдены на рисунках 3 и 4.

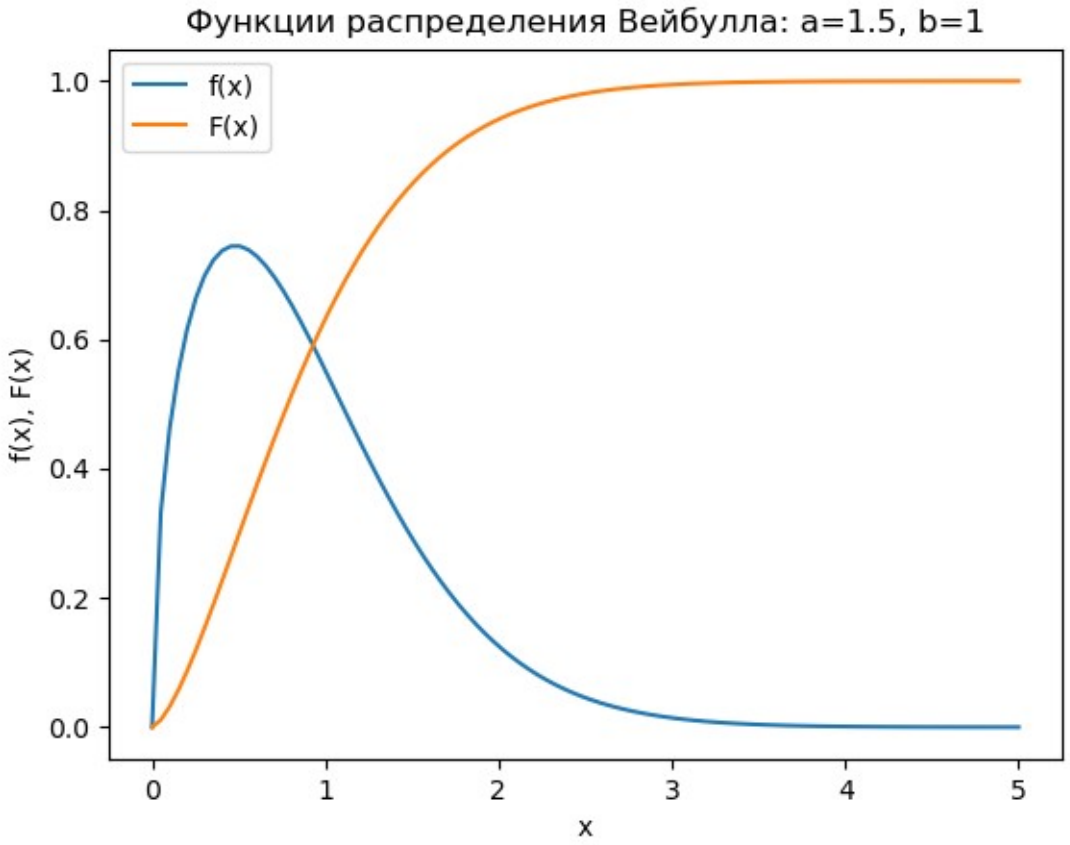

Рисунок 3. Функции распределения Вейбулла для значений параметров: a=1.5, b=1

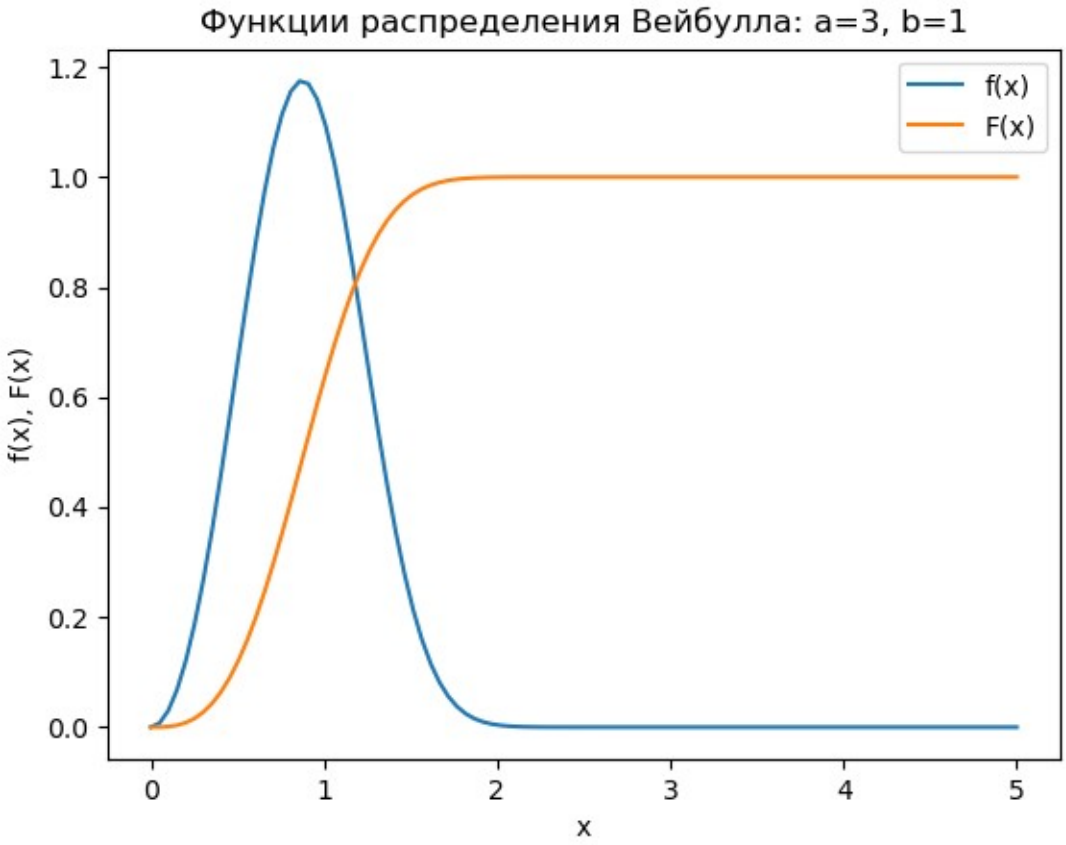

Рисунок 4. Функции распределения Вейбулла для значений параметров: a=3, b=1

С увеличением параметра «a» возрастает островершинность распределения, при этом время безотказной работы изделия будет уменьшаться, и будет уменьшаться также разброс значений долговечности.

Рассмотрим несколько специальных типов распределений, широко применяющихся в математической статистике.

Распределение χ^2 ***("хи-квадрат")***. Пусть $X_i(i=1,2,\dots n)$ - независимые, нормально распределенные случайные величины, причем математическое ожидание каждой из них равно нулю, а среднеквадратическое отклонение – единице (нормированное нормальное распределение). Тогда сумма квадратов этих величин

$$\chi^2=\sum_{i=1}^{n} X_i^2$$

распределена по закону χ^2 («хи квадрат») с $k=n$ степенями свободы. Если же эти величины связаны одним линейным соотношением, например $\sum X_i=n\bar{X}$, то число степеней свободы $k=n-1$.

Плотность этого распределения описывается выражением

$$f(x)=\begin{cases}\dfrac{1}{2^{k/2}\Gamma(k/2)}e^{-x/2}x^{(k/2-1)}, & x>0\\ 0, & x\leqslant 0\end{cases}, \qquad (41)$$

где $\Gamma(x)=\int_0^{\infty} t^{x-1}e^{-t}dt$ - гамма-функция, в частности, для натуральных x

$$\Gamma(n+1)=n! \qquad (42)$$

Из формулы (41) видно, что распределение «хи квадрат» определяется одним параметром – числом степеней свободы k .

В Python для вычисления распределения хи-квадрат и квантилей его используются функции из библиотеки «scipy.stats».

1. Для вычисления функции плотности вероятности (PDF):

- scipy.stats.chi2.pdf(x, df) — возвращает значение функции плотности вероятности для хи-квадрат распределения с df степенями свободы в точке x.

2. Для вычисления интегральной функции распределения (CDF):

- scipy.stats.chi2.cdf(x, df) — возвращает значение интегральной функции распределения для хи-квадрат распределения с df степенями свободы в точке x.

3. Для вычисления квантилей:

- scipy.stats.chi2.ppf(q, df) — возвращает квантиль (обратная функция к CDF) для хи-квадрат распределения с df степенями свободы для вероятности q.

4. Для генерации случайных чисел из хи-квадрат распределения:

- scipy.stats.chi2.rvs(df, size=n) — генерирует n случайных чисел из хи-квадрат распределения с df степенями свободы.

Ниже приведен пример кода для построения графиков функций распределения хи-квадрат.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chi2
# Параметры
df = 10  # степени свободы
x = np.linspace(0, 20, 100)
# Вычисление PDF и CDF
pdf = chi2.pdf(x, df)
cdf = chi2.cdf(x, df)
# Построение графиков
plt.plot(x, pdf, 'b-', label='f(x)')
plt.plot(x, cdf, 'r-', label='F(x)')
plt.title(f'Функции распределения хи-квадрат при df={df}')
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend()
plt.show()
```

Графики функций для различного числа степеней свободы показаны на рисунках 5, 6. С Видно, что с увеличением числа степеней свободы распределение медленно приближается к нормальному.

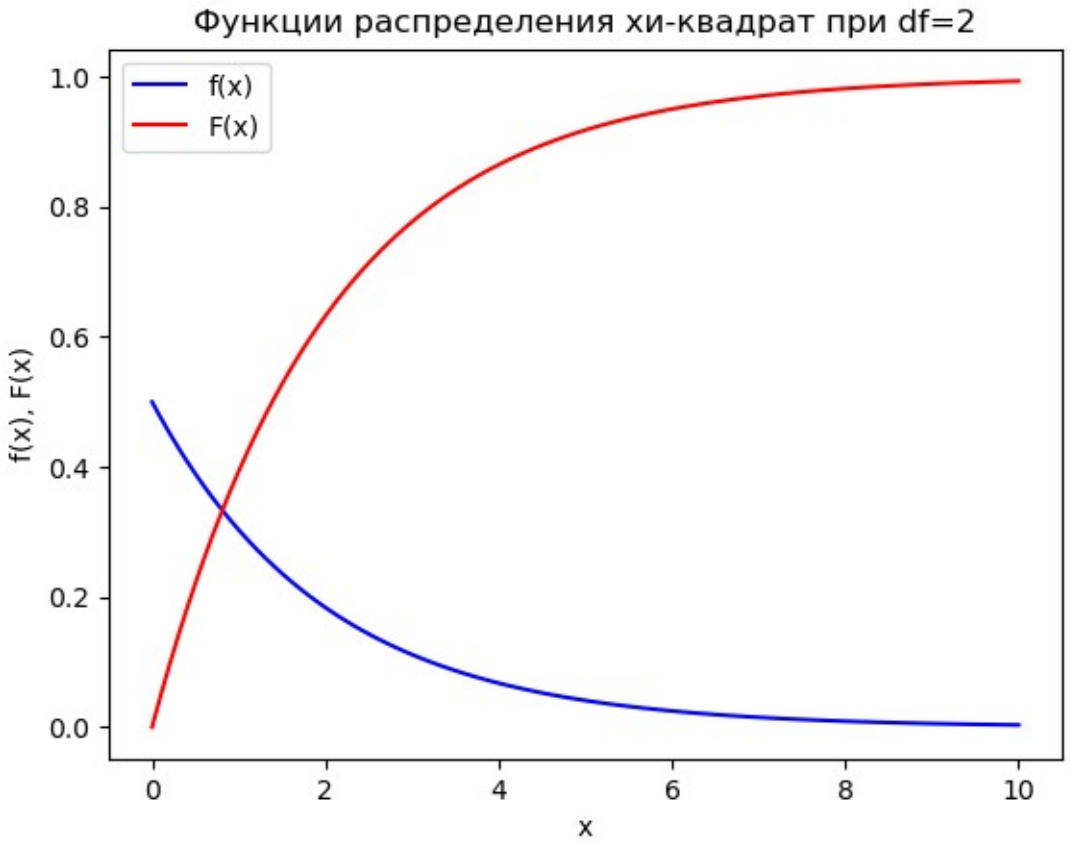

Рисунок 5. Графики функций распределения хи-квадрат для числа степеней свободы df=2

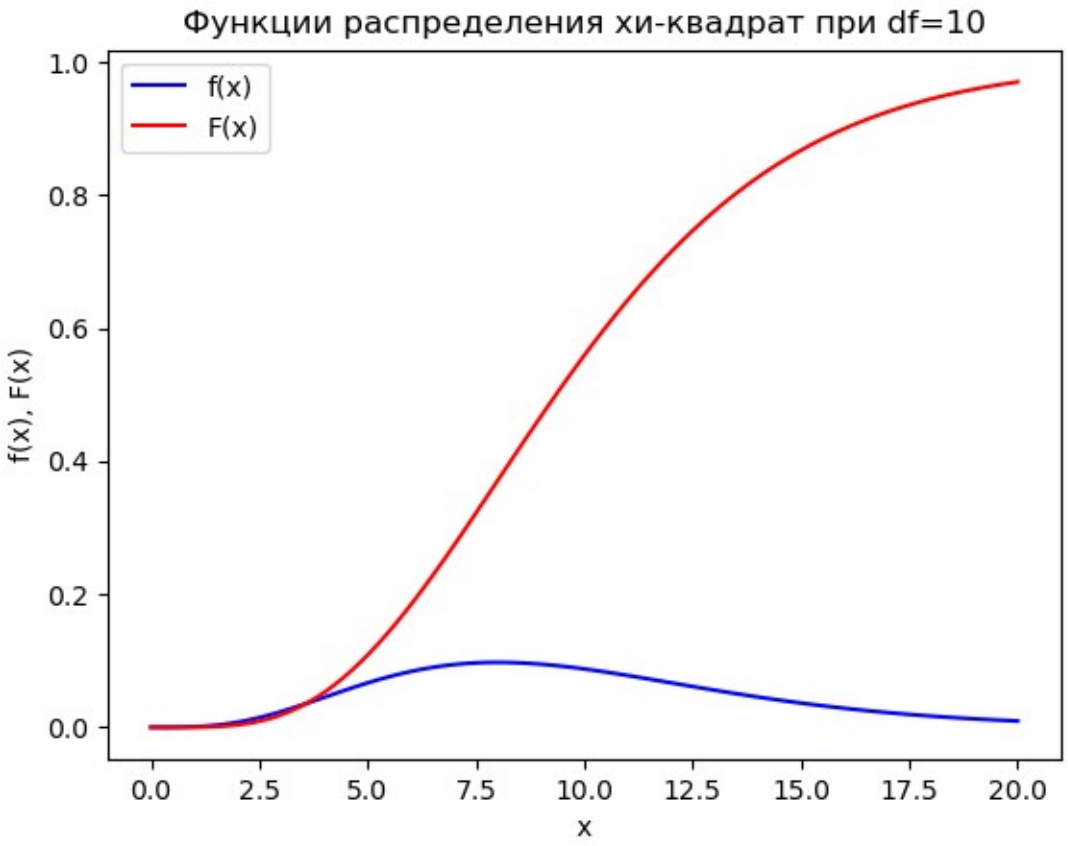

Рисунок 6. Графики функций распределения хи-квадрат для числа степеней свободы df=10

Распределение Стьюдента. Пусть Z - нормально распределенная случайная величина, причем $m_z = 0$, $\sigma_z = 1$, а V - независимая от Z величина, которая распределена по закону χ^2 с k степенями свободы. Тогда величина

$$T = \frac{Z}{\sqrt{V/k}} \tag{43}$$

имеет распределение, которое называют *t*-распределением или распределением Стьюдента (псевдоним английского статистика В. Госсета), с k степенями свободы. Плотность распределения Стьюдента описывается формулой

$$f(x)=\frac{1}{\sqrt{\pi k}}\frac{\Gamma\left(\frac{k+1}{2}\right)}{\Gamma\left(\frac{k}{2}\right)}\left(1+\frac{x^2}{k}\right)^{\frac{k+1}{2}} \quad . \qquad (44)$$

В Python для работы с распределением Стьюдента и его квантилями используются функции из библиотеки scipy.stats.

1. Для вычисления функции плотности вероятности (PDF):

- scipy.stats.t.pdf(x, df) — возвращает значение функции плотности вероятности для распределения Стьюдента с df степенями свободы в точке x.

2. Для вычисления интегральной функции распределения (CDF):

- scipy.stats.t.cdf(x, df) — возвращает значение интегральной функции распределения для распределения Стьюдента с df степенями свободы в точке x.

3. Для вычисления квантилей:

- scipy.stats.t.ppf(q, df) — возвращает квантиль (обратная функция к CDF) для распределения Стьюдента с df степенями свободы для вероятности q.

4. Для генерации случайных чисел из распределения Стьюдента:

- scipy.stats.t.rvs(df, size=n) — генерирует n случайных чисел из распределения Стьюдента с df степенями свободы.

Ниже приведен код для построения графиков функций распределения Стьюдента.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t
# Параметры
```

```
df = 5  # степени свободы
x = np.linspace(-5, 5, 100)
# Вычисление PDF и CDF
pdf = t.pdf(x, df)
cdf = t.cdf(x, df)
# Построение графиков
plt.plot(x, pdf, 'b-', label='f(x)')
plt.plot(x, cdf, 'r-', label='F(x)')
plt.title(f'Функции распределения Стьюдента при df={df}')
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend()
plt.show()
```

Графики дифференциальной и интегральной функций распределения Стьюдента показаны на рисунке 7.

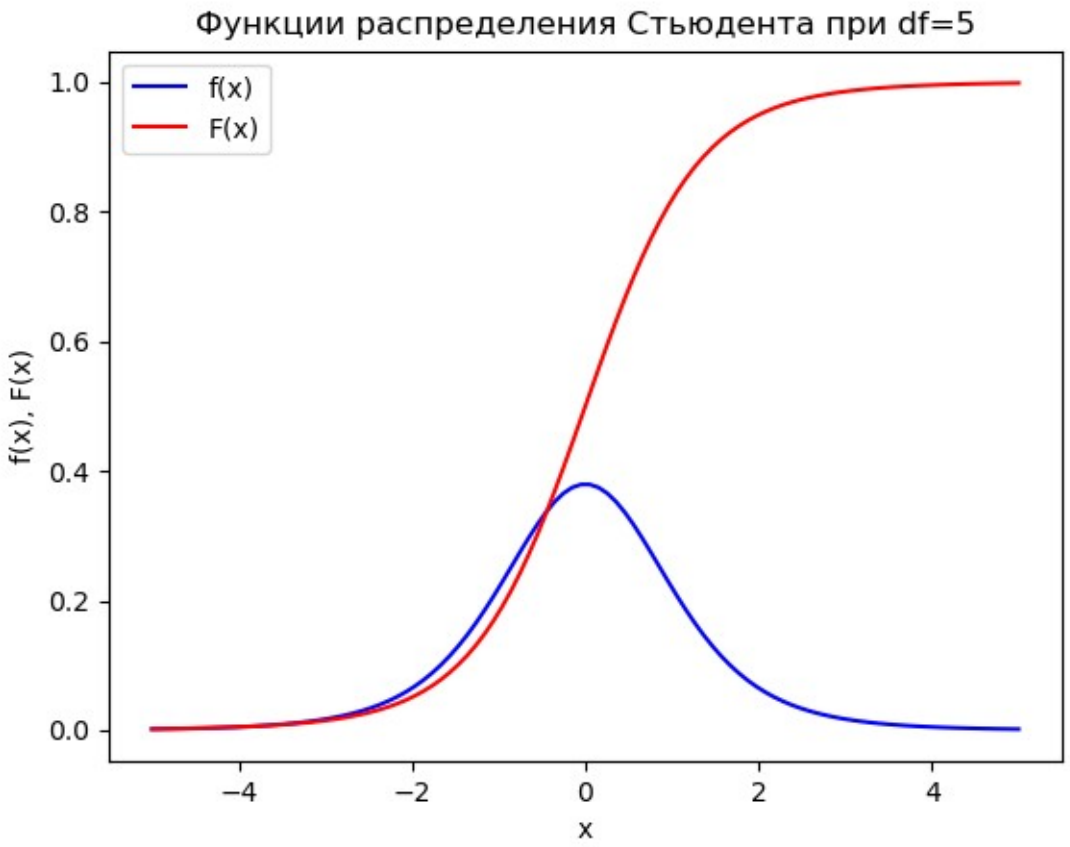

Рисунок 7. Графики функций распределения Стьюдента для числа степеней свободы df=5

Видно, что распределение Стьюдента, как и стандартное нормальное распределение, симметрично относительно точки $x=0$. С возрастанием числа степеней свободы распределение Стьюдента быстро приближается к нормальному.

Распределение Фишера. Если *U* и *V* – независимые случайные величины, распределенные по закону χ^2 со степенями свободы k_1 и k_2 соответственно, то величина

$$F=\frac{U/k_1}{V/k_2} \tag{45}$$

имеет распределение, которое называют распределением Фишера (Фишера-Снедекора) со степенями свободы k_1 и k_2 .

Плотность этого распределения описывается функцией

$$f(x)=\begin{cases} C_0\dfrac{x^{(k_1-2)/2}}{(k_2+k_1x)^{(k_1+k_2)/2}}, x>0 \\ 0, \qquad x\leqslant 0 \end{cases} \tag{46}$$

где

$$C_0=\frac{\Gamma\left(\dfrac{k_1+k_2}{2}\right)k_1^{k_1/2}k_2^{k_2/2}}{\Gamma(k_1/2)\Gamma(k_2/2)} \quad .$$

Как видно, распределение *F* определяется двумя параметрами – числами степеней свободы k_1 , k_2 .

В Python для работы с распределением Фишера и его квантилями используются функции из библиотеки scipy.stats.

1. Для вычисления функции плотности вероятности (PDF):

- scipy.stats.f.pdf(x, dfn, dfd) — возвращает значение функции плотности вероятности для распределения Фишера с dfn и dfd степенями свободы в точке x.

2. Для вычисления интегральной функции распределения (CDF):

- scipy.stats.f.cdf(x, dfn, dfd) — возвращает значение интегральной функции распределения для распределения Фишера с dfn и dfd степенями свободы в точке x.

3. Для вычисления квантилей:

- scipy.stats.f.ppf(q, dfn, dfd) — возвращает квантиль (обратная функция к CDF) для распределения Фишера с dfn и dfd степенями свободы для вероятности q.

4. Для генерации случайных чисел из распределения Фишера:

- scipy.stats.f.rvs(dfn, dfd, size=n) — генерирует n случайных чисел из распределения Фишера с dfn и dfd степенями свободы.

Ниже приведен код для построения графиков функций распределения Фишера для заданных значений степеней свободы.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import f
# Параметры
dfn = 5  # степени свободы в числителе
dfd = 2  # степени свободы в знаменателе
x = np.linspace(0, 5, 100)
# Вычисление PDF и CDF
pdf = f.pdf(x, dfn, dfd)
cdf = f.cdf(x, dfn, dfd)
# Построение графиков
plt.plot(x, pdf, 'b-', label='f(x)')
plt.plot(x, cdf, 'r-', label='F(x)')
plt.title(f'Функции распределения Фишера при dfn={dfn} и dfn={dfd}')
plt.xlabel('x')
plt.ylabel('f(x), F(x)')
plt.legend()
plt.show()
```

Графики функций распределения Фишера показаны на рисунке 8. Как видно, данное распределение является сильно асимметричным.

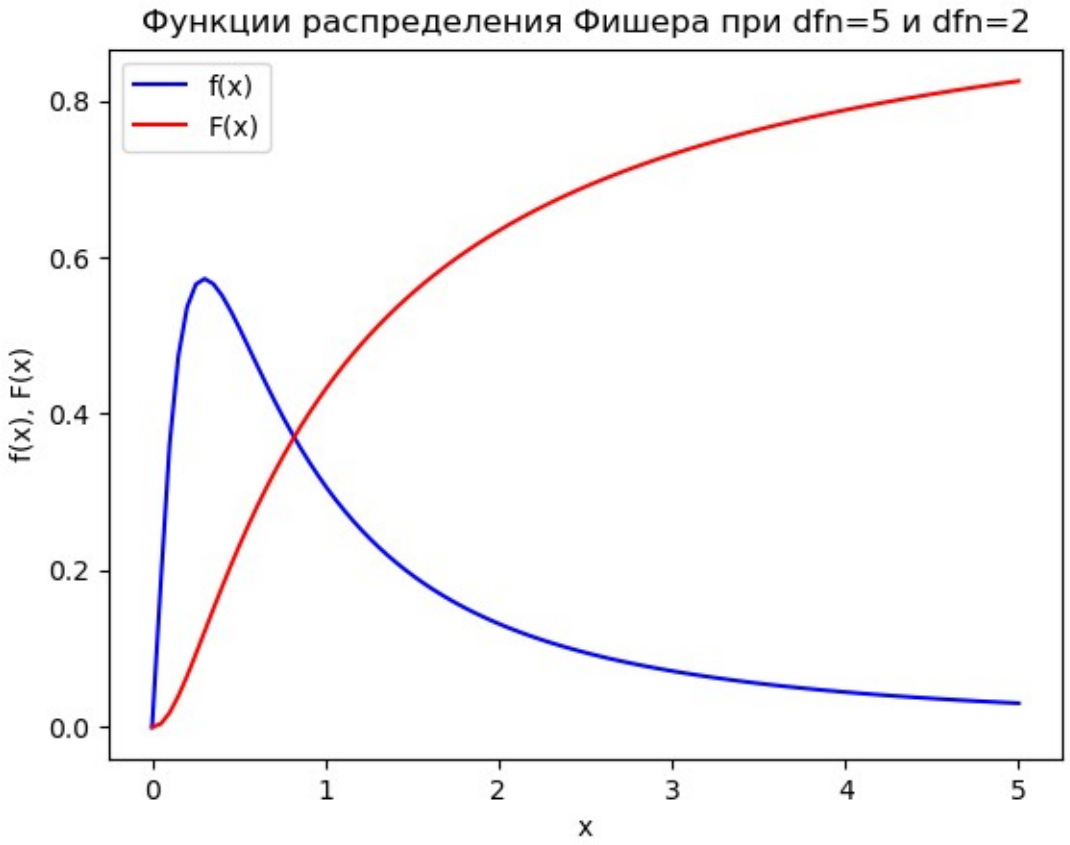

Рисунок 8. Графики дифференциальной и интегральной функций распределения Фишера

2.3.2. Многомерные случайные величины

Зачастую для описания практической ситуации оказывается необходимым использование одновременно нескольких (в простейшем случае — двух) случайных величин. Для задания вероятностных свойств двух случайных величин *X*, *Y* используются двумерные (совместные) функции распределения вероятностей: интегральная $F(x,y)$ и дифференциальная $f(x,y)$. Функция $F(x,y)$, характеризующая вероятность того, что первая случайная величина принимает некоторое значение, меньшее или равное *x*, а вторая—значение, меньшее или равное *y*, называется *интегральной функцией* совместного распределения двух случайных величин.

$$F(x,y)=P\{X\le x;Y\le y\}\ . \qquad (47)$$

Как и для одной непрерывной случайной величины, если функция $F(x,y)$ достаточно гладкая, то ее можно продифференцировать, в результате чего получится *двумерная дифференциальная функция* распределения вероятностей (двумерная плотность вероятности)

$$f(x,y)=\frac{\partial^2}{\partial x\,\partial y}F(x,y)\ . \qquad (48)$$

Функция *f(x, y)* обладает следующими свойствами:

1) $f(x,y)\geq 0$; 2) $\int\limits_{-\infty}^{\infty}\int f(x,y)dxdy=1$;

3) $F(x,y)=\int\limits_{-\infty}^{x}\int\limits_{-\infty}^{y} f(z_1,z_2)dz_1dz_2$; 4) $\lim\limits_{\substack{|x|\to\infty\\|y|\to\infty}} f(x,y)=0$

(z_1 , z_2 —переменные интегрирования).

Вероятность того, что случайные величины *X, У* одновременно попадут в некоторую произвольную область Ω,, составляет

$$P\{(X,Y)\in\Omega\}=\iint\limits_{\Omega} f(x,y)\,dxdy\ . \qquad (49)$$

В частности,

$$P\{a_1<X<a_2;\,b_1<Y<b_2\}=\int\limits_{a_1}^{a_2}\int\limits_{b_1}^{b_2} f(x,y)\,dxdy\ . \qquad (50)$$

По известной двумерной плотности $f(x,y)$ легко найти *частные (одномерные) функции распределения* $f(x)$, $f(y)$ каждой случайной величины

$$f(x)=\int\limits_{-\infty}^{\infty} f(x,y)\,dy\ ,\ f(y)=\int\limits_{-\infty}^{\infty} f(x,y)\,dx\ . \qquad (51)$$

Две случайные величины *X* и *Y* называются *независимыми*, если

$$f(x,y)=f(x)f(y) \quad . \tag{52}$$

Как и в одномерном случае, основные свойства двумерной совокупности величин *X, Y* могут быть охарактеризованы с помощью ряда числовых параметров. При этом в качестве наиболее употребительных параметров, описывающих поведение каждой из случайных величин в отдельности, как и выше, используются математическое ожидание и дисперсия соответствующей случайной величины: m_x m_y σ_x^2 σ_y^2. Кроме подобного рода параметров для двумерной совокупности могут быть построены параметры, характеризующие степень взаимозависимости переменных *X* и *У.* Простейшими из них является *ковариация* двух случайных величин (называемая также *корреляционным моментом)*

$$cov(X,Y)=\int_{-\infty}^{\infty}\int (x-m_x)(y-m_y)f(x,y)\,dxdy \quad , \tag{53}$$

а также нормированный показатель связи — *коэффициент корреляции*

$$\rho_{xy}=\frac{\mathrm{cov}(X,Y)}{\sigma_x\sigma_y} \quad . \tag{54}$$

По своему смыслу коэффициент корреляции является далеко не исчерпывающей характеристикой статистической связи, характеризуя лишь степень линейной зависимости между *X* и *Y.* Коэффициент корреляции меняется в пределах $-1\le\rho_{xy}\le1$. Если $\rho_{xy}=1$, то случайные величины полностью положительно коррелированы, т. е. $y=a_0+a_1x$, где a_0 и a_1 — постоянные, причем $a_1>0$. Если же $\rho_{xy}=-1$, то случайные величины полностью отрицательно коррелированы, т. е. $y=a_0-a_1x$. Если $\rho_{xy}=0$, то говорят, что случайные величины *X* и *Y* не коррелированы: $a_1=0$. В том случае, когда *X* и *Y* — независимые случайные величины, для них $\rho_{xy}=0$; следовательно, они и не коррелированы. Обратное утверждение в общем случае неверно: *X* и *Y* могут быть связаны даже функционально и все же иметь нулевой коэффициент корреляции (при этом, конечно, функциональная связь должна быть нелинейной).

В качестве примера функции распределения двумерной случайной величины рассмотрим нормальный закон распределения на плоскости. Плотность двумерной случайной величины, распределенной по нормальному закону, описывается функцией

$$f(x,y)=\frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho_{xy}^2}}\exp\left\{-\frac{1}{2\left(1-\rho_{xy}^2\right)}\left[\frac{\left(x-m_x\right)^2}{\sigma_x^2}+\frac{\left(y-m_y\right)^2}{\sigma_y^2}-2\rho_{xy}\frac{x-m_x}{\sigma_x}\frac{y-m_y}{\sigma_y}\right]\right\}$$

Мы видим, что нормальный закон на плоскости описывается пятью параметрами: двумя математическими ожиданиями m_x , m_y ; двумя среднеквадратическими отклонениями σ_x , σ_y и коэффициентом корреляции ρ_{xy} .

В общем случае $k>1$ случайных величин $X_1, X_2, \dots X_k$ удобно рассматривать их как компоненты случайного вектора, т.е. упорядоченного набора из чисел, расположенных в виде столбца

$$\mathbf{X}=\begin{pmatrix} X_1 \\ X_2 \\ . \\ . \\ X_k \end{pmatrix}$$

Реализация случайного вектора обозначается вектором наблюдений

$x=\left(x_1, x_2 \dots x_k\right)^T$,

где компоненты вектора x являются реализациями $x_1, x_2 \dots x_k$ случайных величин $X_1, X_2, \dots X_k$ соответственно. Такая реализация называется многомерным наблюдением.

Основной характеристикой многомерной случайной величины является ее функция распределения

$$F\left(x_1, x_2, \dots x_k\right)=P\left(X_1<x_1, X_2<x_2, \dots X_k<x_k\right) \quad . \qquad (55)$$

Аналогично с одномерными случайными величинами для многомерной случайной величины можно ввести также *дифференциальную функцию распределения* $f(x_1, x_2, \dots x_k)$ (плотность распределения)

$$F(x_1, x_2, \dots x_k) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \dots \int_{-\infty}^{x_k} f(t_1, t_2 \dots t_k)\, dt_1 dt_2 dt_k \qquad (56)$$

Связь между интегральной и дифференциальной функцией распределения можно записать также в виде

$$\frac{\partial^k F(x_1, x_2, \dots x_k)}{\partial x_1 \dots \partial x_k} = f(x_1, \dots x_k) \quad . \qquad (57)$$

Так же как и для одномерной плотности распределения, интеграл от плотности многомерного распределения по всей области определения равен единице:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} f(x_1, \dots x_k)\, dx_1 \dots dx_k = 1 \quad ,$$

поскольку представляет собой вероятность достоверного события.

Пусть $m < k$, тогда

$$f(x_1, \dots x_m) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} f(x_1, \dots x_m, \dots x_k)\, dx_{m+1} \dots dx_k \quad . \qquad (58)$$

В частности,

$$f(x_i) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} f(x_1, \dots x_i, \dots x_k)\, dx_1 \dots dx_{i-1} dx_{i+1} \dots dx_k \quad . \qquad (59)$$

В случае, когда случайные величины $X_1,\dots X_k$ независимые,

$$F(x_1,\dots x_k)=F(x_1)F(x_2)\dots F(x_k)$$

и

$$f(x_1,\dots x_k)=f(x_1)f(x_2)\dots f(x_k) \quad .$$

Для многомерных распределений могут быть вычислены параметры, аналогичные параметрам двумерных распределений.

Математическое ожидание *i*-й компоненты случайного вектора

$$m_i=E(X_i)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\dots\int_{-\infty}^{\infty}x_i f(x_1,\dots x_k)\,dx_1\dots dx_k \quad . \qquad (60)$$

Дисперсия *i*-й компоненты случайного вектора

$$\sigma_i^2=E((X_i-m_i)^2)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\dots\int_{-\infty}^{\infty}(x_i-m_i)^2 f(x_1,\dots x_k)\,dx_1\dots dx_k \quad . \qquad (61)$$

Ковариация *i*-й и *j*-й компонент случайного вектора

$$\sigma_{ij}=E((X_i-m_i)(X_j-m_j))=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\dots\int_{-\infty}^{\infty}(x_i-m_i)(x_j-m_j)f(x_1,\dots x_k)\,dx_1\dots dx_k \quad .$$

Компоненты математических ожиданий образуют вектор m математического ожидания

$m=E(X)=(m_1,m_2,\dots m_k)^T$, а ковариации и дисперсии образуют вместе ковариационную матрицу Σ :

$$\Sigma = \mathrm{cov}(X) = \begin{pmatrix} \sigma_1^2 \sigma_{12} \ldots \sigma_{1k} \\ \sigma_{21} \sigma_2^2 \ldots \sigma_{2k} \\ \ldots\ldots\ldots\ldots \\ \sigma_{k1} \sigma_{k2} \ldots \sigma_k^2 \end{pmatrix}. \qquad (62)$$

Из определения ковариации следует, что ковариационная матрица симметрична.

В задачах практики часто приходится определять распределение одних компонентов случайного вектора при известных значениях других – так называемые условные распределения.

Пусть $x = (x_1, x_2, \ldots x_k)^T$ - *k*-компонентный случайный вектор. Обозначим посредством $z = (z_1, z_2, \ldots z_m)^T$, $m<k$ вектор, компоненты которого представляют собой первые *m* компонентов вектора x, а $y = (y_1, y_2 \ldots y_l)^T$, $l = k - m$ – вектор, компоненты которого представляют собой оставшиеся *l* компонентов вектора x. Тогда условная плотность случайной величины Z при заданном значении y случайной величины Y выражается следующим образом:

$$f(z|y) = \frac{f(z, y)}{f(y)}, \qquad (63)$$

где $f(z, y) \equiv f(x)$, а плотность распределения $f(y)$ выражается в соответствии с (58) как

$$f(y) = \int_{-\infty}^{\infty} f(z, y)\, dz = \int_{-\infty}^{\infty} \ldots \int_{-\infty}^{\infty} f(x_1, x_2, \ldots x_m, \ldots x_k)\, dx_1\, dx_2 \ldots dx_m. \qquad (64)$$

Из многомерных распределений в статистических приложениях чаще всего используется многомерное нормальное распределение. Оно задается вектором средних значений m и матрицей ковариации Σ . Дадим формальное определение многомерного нормального распределения.

Пусть $Z_1, Z_2, \ldots Z_k$ - взаимно независимые случайные величины, каждая из которых распределена по одномерному нормальному закону с нулевым математическим ожиданием и единичной дисперсией. Тогда вектор $Z=(Z_1, Z_2, \ldots Z_k)^T$ обладает *стандартным сферическим нормальным распределением* с плотностью

$$f(z)=\frac{1}{(2\pi)^{k/2}} e^{-(1/2) z^T z} \quad , \qquad (65)$$

где $z=(z_1, z_2, \ldots z_k)^T$.

Обозначим распределение Z через $N(0, I)$, где 0 – нулевой вектор, а I – единичная матрица. Если A – произвольная невырожденная матрица констант, размерности $k \times k$, m – *k*-мерный вектор констант, то $X=\mathbf{AZ}+m$ обладает *многомерным* (или *k*-мерным) *невырожденным нормальным распределением*. Его плотность имеет вид

$$f(x)=(2\pi)^{-k/2} |\Sigma|^{-1/2} \exp\left[-\frac{1}{2}(x-m)^T \Sigma^{-1}(x-m)\right] \quad , \qquad (66)$$

где $x=(x_1, x_2, \ldots x_k)^T$, $\Sigma=\mathbf{AA}^{\boldsymbol{T}}$, $|\Sigma|$ - определитель матрицы Σ , а Σ^{-1} - матрица, обратная Σ . Вектор математического ожидания этого распределения равен m , а матрица ковариации равна Σ . В этом случае говорят, что случайный вектор X распределен

по закону $N(m,\Sigma)$. Область *k*-мерного евклидова пространства, определяемая уравнением $f(x)=c$, где с – константа, является эллипсоидом, называемым *эллипсоидом концентрации*.

Если *k*-мерный вектор x имеет распределение $N(m,\Sigma)$, а B – матрица $m\times k$ ранга *m*, то *m*-мерный вектор $y=Bx$ обладает *m*-мерным нормальным распределением.

Пусть случайный вектор $X=(X_1,X_2,\dots X_k)^T$ имеет распределение $N(m,\Sigma)$ и $X_1=(X_1,X_2\dots X_l)^T$, $X_2=(X_{l+1},X_{l+2}\dots X_k)^T$. Кроме того, положим $m_1=(m_1,m_2\dots m_l)^T$ и $m_2=(m_{l+1},m_{l+2}\dots m_k)^T$,

$$\Sigma_{11}=\begin{pmatrix}\sigma_1^2\sigma_{12}\dots\sigma_{1l}\\ \sigma_{21}\sigma_2^2\dots\sigma_{2l}\\ \dots\dots\dots\dots\\ \sigma_{l1}\sigma_{l2}\dots\sigma_l^2\end{pmatrix},\quad \Sigma_{22}=\begin{pmatrix}\sigma_{l+1}^2\sigma_{l+1,2}\dots\sigma_{l+1,k}\\ \sigma_{l+2,l+1}\sigma_{l+2}^2\dots\sigma_{l+2,k}\\ \dots\dots\dots\dots\\ \sigma_{k,l+1}\sigma_{k,l+2}\dots\sigma_k^2\end{pmatrix},\quad \Sigma_{12}=\Sigma_{21}^T=\begin{pmatrix}\sigma_{1,l+1}\sigma_{1,l+2}\dots\sigma_{1k}\\ \sigma_{2,l+1}\sigma_{2,l+2}\dots\sigma_{2k}\\ \dots\dots\dots\dots\\ \sigma_{l,l+1}\sigma_{l,l+2}\dots\sigma_{lk}\end{pmatrix}.$$

Тогда X_1 имеет распределение $N(m_1,\Sigma_{11})$, а X_2 - распределение $N(m_2,\Sigma_{22})$. Условным распределением X_1 при условии $X_2=x_2=(x_{l+1},\dots x_k)^T$ будет $N\left(m_1+\Sigma_{12}\Sigma_{22}^{-1}(x_2-m_2),\Sigma_{11}-\Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}\right)$.

Для вычисления многомерного нормального распределения в Python используются функции из библиотеки scipy.stats.

1. Функция плотности вероятности (PDF):

- scipy.stats.multivariate_normal.pdf(x, mean, cov) — возвращает значение функции плотности вероятности для многомерного нормального распределения. Здесь x — это массив значений (многомерный), mean — среднее значение (вектор), а cov — матрица ковариации.

2. Интегральная функция распределения (CDF):

- scipy.stats.multivariate_normal.cdf(x, mean, cov) — возвращает значение интегральной функции распределения для многомерного нормального распределения. Параметры аналогичны функции PDF.

3. Генерация случайных чисел:

- scipy.stats.multivariate_normal.rvs(mean, cov, size=n) — генерирует массив из n случайных чисел из многомерного нормального распределения с заданным средним и матрицей ковариации.

В качестве примера приведем код программы для построения графика функции плотности вероятности двумерного нормального распределения.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal
# Задаем параметры двумерного нормального распределения
mean = [0, 0]  # Среднее значение
cov = [[1, 0], [0, 1]]  # Ковариационная матрица
# Создаем сетку для X и Y
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
# Вычисляем плотность вероятности
pos = np.dstack((X, Y))
rv = multivariate_normal(mean, cov)
Z = rv.pdf(pos)
# Создаем 3D график
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
# Строим поверхность
ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
# Настраиваем график
```

```
ax.set_title('Плотность вероятности двумерного нормального распределения')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Плотность вероятности')
# Показать график
plt.show()
```

Пример графика показан на рисунке 9.

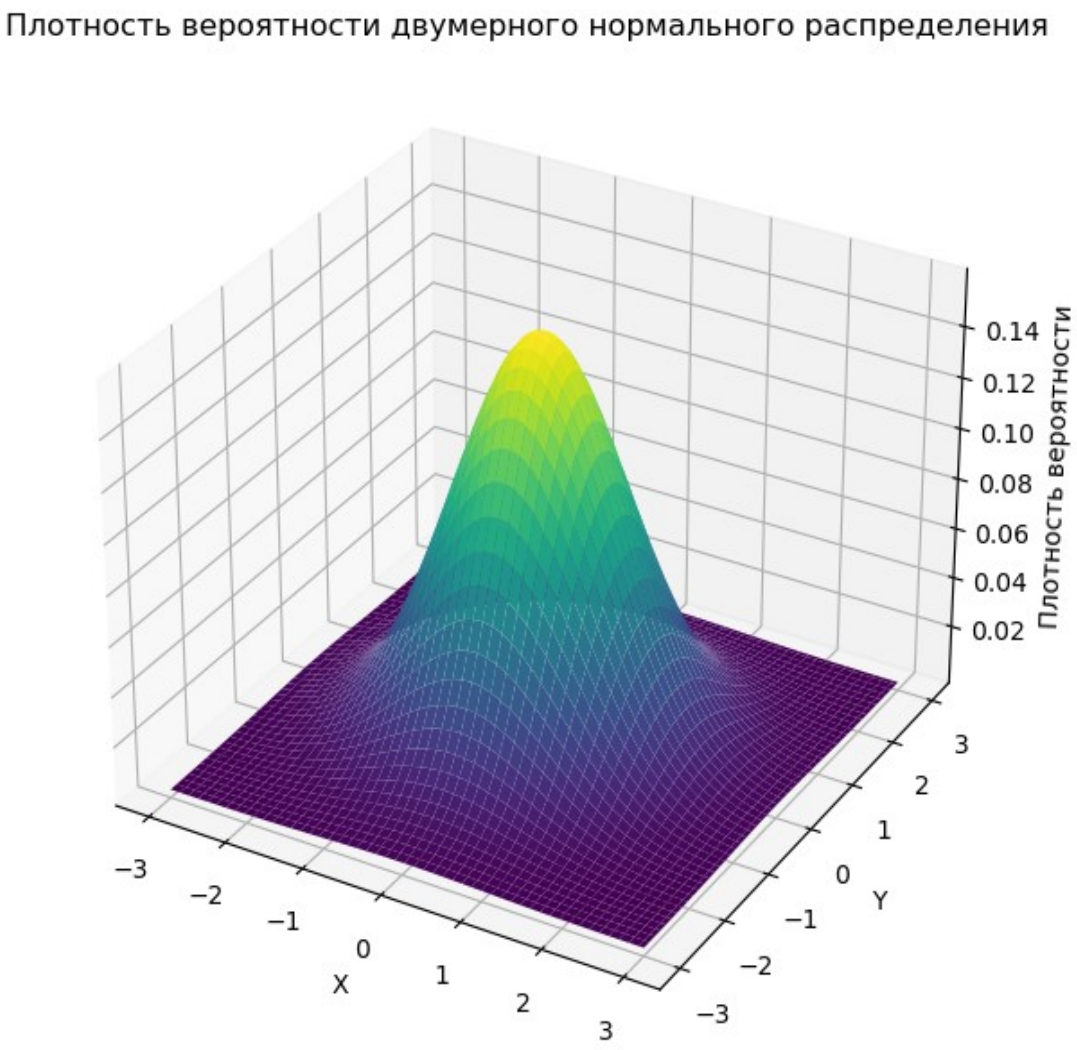

Рисунок 9. График плотности вероятности двумерного нормированного нормального распределения

Глава 3. Статистический анализ и планирование эксперимента

3.1. Экспериментальный анализ параметров распределений случайных величин

Все описанные выше функции и связанные с ними параметры являются теоретическими, характеризующими определенные свойства изучаемого объекта. На практике почти всегда эти характеристики неизвестны и возникает задача экспериментального (эмпирического) определения тех или иных характеристик случайных величин на основе наблюдений.

Фундаментальными понятиями статистической теории являются понятия генеральной совокупности и выборки.

Генеральная совокупность обычно интерпретируется как совокупность всех мыслимых (возможных) результатов наблюдений над случайной величиной, которые в принципе могут быть проведены при данных условиях.

Содержательный смысл этого понятия состоит в том, что предполагается существование некоторых вполне определенных свойств, неслучайных закономерностей, присущих данной совокупности, тех свойств, которые и должны быть выяснены исследователем. Фактически эти свойства являются объективным отображением вероятностных свойств изучаемого объекта, которые могут быть охарактеризованы с помощью соответствующих законов распределения вероятностей или связанных с ними числовых параметров.

Считается, что указанные свойства не изменяются во времени и присущие генеральной совокупности неслучайные закономерности сохраняют постоянным свой характер, т. е. являются *устойчивыми.*

Выборка — это конечный набор значений случайной величины, полученный в результате наблюдений. Число элементов выборки называется ее *объемом.* Если, например, x_1, x_2, ..., x_N — наблюдаемые значения случайной величины X (возможно, и совпадающие), то объем данной выборки равен N.

Выборка называется *репрезентативной (представительной),* если она достаточно полно характеризует генеральную совокупность. Для обеспечения репрезентативности выборки чаще всего используется случайный выбор элементов. Предполагается, что при таком выборе каждая

возможная выборка фиксированного объема имеет одну и ту же вероятность выбора, а последовательные наблюдения взаимно независимы.

Смысл статистических методов заключается в том, чтобы по выборке ограниченного объема *N*, т. е. по некоторой части генеральной совокупности, высказать обоснованное суждение о ее свойствах в целом. Подобное суждение может быть получено путем построения эмпирических (выборочных) аналогов вероятностных характеристик исследуемой величины, иначе говоря, путем оценивания параметров (характеристик) генеральной совокупности с помощью некоторых подходящих функций от результатов наблюдений — *оценок.*

При многократном извлечении выборок одного и того же объема и последующем нахождении множества оценок одного и того же параметра получаются различные числовые значения этих оценок, изменяющиеся от одной выборки к другой случайным образом. Иными словами, любая оценка произвольного параметра Θ есть случайная величина.

В этом ее принципиальное отличие от самого оцениваемого параметра Θ, являющегося неслучайным. Чтобы подчеркнуть указанное существенное обстоятельство, для параметров генеральной совокупности и их оценок вводятся разные обозначения: в общем случае оценка произвольного параметра Θ обозначается через $\hat{\Theta}$. Оценка математического ожидания (генерального среднего значения) m_x, обозначается чаще всего через $\overline{x}$, оценка дисперсии σ_x^2 — через s_x^2 или просто s^2. Естественно, что вероятностные свойства произвольной оценки $\hat{\Theta}=\hat{\Theta}(x_1, x_2, \ldots x_N)$ параметра Θ можно описать с помощью функции распределения оценки $f_{\hat{\Theta}}(\hat{\Theta})$ или ее характеристик $m_{\hat{\Theta}}, \sigma^2_{\hat{\Theta}}$.

Для оценивания одного и того же параметра можно использовать в принципе различные оценки. Чтобы выбрать наилучшую из них, необходимо сформулировать некоторые требования к свойствам оценок, желательные с точки зрения практики.

Оценка $\hat{\Theta}$ параметра Θ называется *состоятельной,* если при неограниченном увеличении объема выборки *N* значение $\hat{\Theta}$ с полной мерой достоверности (с вероятностью

единица) стремится к своему теоретическому значению Θ. Это означает, что с ростом *N* распределение $f_{\hat{\Theta}}(\hat{\Theta})$ все в большей степени концентрируется вокруг Θ. Состоятельность оценки гарантирует исследователю увеличение точности оценивания с ростом *N* и то, что хотя бы в пределе при $N \to \infty$ он может получить точное значение Θ.

Оценка $\hat{\Theta}$ называется *несмещенной,* если $m_{\hat{\Theta}} = \Theta$ для любого *N*.

Несмещенность означает отсутствие систематической погрешности при оценивании параметра Θ.

Оценка $\hat{\Theta}$ называется *эффективной,* если среди всех оценок параметра Θ она обладает наименьшей дисперсией $\sigma^2_{\hat{\Theta}}$. Эффективная оценка, следовательно, имеет минимальную случайную ошибку и в этом смысле является наиболее точной.

Свойства оценок различных параметров Θ во многом определяются видом закона распределения исследуемой генеральной совокупности. В практических расчетах часто априори предполагается, что этот закон является нормальным. Однако в действительности это не всегда так, и установление вида закона распределения исследуемой случайной величины является важным элементом статистического анализа.

При рассмотрении примеров применения различных статистических методов в иллюстративных целях необходимо иметь выборки случайных величин, полученные на основании эксперимента. В практических руководствах по математической статистике они приводятся в качестве исходных данных в примерах решения задач, например: " В результате измерения температуры в реакторе получена следующая выборка экспериментальных значений …". Мы не будем приводить подобных выборок, поскольку конкретная физическая природа экспериментальных данных при изучении статистических методов не играет никакой роли. Вместо этого при рассмотрении примеров мы будем *моделировать* случайные величины. Для этой цели мы будем использовать рассмотренные в предыдущей главе функции генерирования псевдослучайных чисел с различными распределениями, реализованные в библиотеке scipy.stats.

3.1.1. Одномерные случайные величины

Обработку экспериментальных данных обычно производят в следующем порядке.

1. Построение вариационного ряда. Пусть имеется выборка экспериментальных данных $x_1, x_2, \ldots, x_N$. Вариационный ряд $z_1, z_2, \ldots z_N$ получают из исходных данных путем расположения x_m $(m = 1,2, \ldots N)$ в порядке возрастания от x_{min} до x_{max} так, чтобы $x_{min}=z_1 \le z_2 \le \ldots \le z_N=x_{max}$.

2. Построение гистограммы распределения. Гистограмма $\hat{f}(x)$ является эмпирическим аналогом функции плотности распределения $f(x)$. Обычно ее строят следующим образом:

1. Весь вариационный ряд случайной величины разбивают на интервалы. Для этого находят предварительное количество интервалов, на которое должна быть разбита ось Ox. Это количество K определяют с помощью оценочной формулы

$$K = 1 + 3.2 \lg N \quad , \tag{67}$$

где найденное значение округляют до ближайшего целого числа.

2. Определяют длину интервала

$$\Delta x = (x_{\max} - x_{\min}) / K \quad . \tag{68}$$

Величину Δx можно несколько округлить для удобства вычислений.

3. Середину области изменения выборки (центр распределения) $(x_{\max} + x_{\min})/2$ принимают за центр некоторого интервала, после чего легко находят границы и окончательное количество указанных интервалов так, чтобы в совокупности они перекрывали всю область от $x_{\min}$ до $x_{\max}$.

4. Подсчитывают количество наблюдений N_m, попавшее в каждый интервал: N_m равно числу членов вариационного ряда, для которых справедливо неравенство

$$x_m \le z_l < x_m + \Delta x \quad . \tag{69}$$

Здесь x_m и $x_m + \Delta x$ - границы m-го интервала. Отметим, что при использовании формулы (68) значения z_l, попавшие на границу между (m-1) и m-м интервалами, относят к m-му интервалу.

5. Подсчитывают относительное количество (относительную частоту) наблюдений N_m/N, попавших в данный интервал.

6. Строят гистограмму, представляющую собой ступенчатую кривую, значение которой на m-м интервале $(x_m,\ x_m+\Delta x)$ (m=1, 2, ..., K) постоянно и равно N_m/N, или с учетом условия $\int_{-\infty}^{\infty} \hat{f}(x)\,dx = 1$ равно $N_m/(N\Delta x)$.

Для неодинаковых интервалов оценку плотности распределения на m-м интервале находят по формуле

$$\hat{f}(x_m) = \frac{N_m}{N \cdot \Delta x_m} \quad . \tag{70}$$

3. Построение диаграммы накопленных частот $\hat{F}(x)$, являющейся эмпирическим аналогом интегральной функции распределения. Диаграмму строят в соответствии с формулой

$$\hat{F}(x_m) = \sum_{j=1}^{m} \hat{f}(x_j)\,\Delta x_j \quad . \tag{71}$$

4. Определение оценок математического ожидания $\overline{x}$ **, дисперсии** s_x^2 **и среднего квадратического отклонения** s_x . Оценкой математического ожидания является среднее значение выборки

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \quad . \tag{72}$$

Оценку дисперсии и среднеквадратического отклонения вычисляют по формулам

$$s_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(x_i - \overline{x}\right)^2 \quad , \tag{73}$$

$$s_x = +\sqrt{s_x^2} \quad . \tag{74}$$

Оценки коэффициентов асимметрии ($\hat{A}s$) и эксцесса ($\hat{E}k$) вычисляются соответственно по формулам

$$\hat{A}s = \frac{N}{(N-1)(N-2)}\sum_{i=1}^{N}\left(\frac{x_i - \overline{x}}{s_x}\right)^3 \quad , \tag{75}$$

$$\hat{E}k = \frac{N(N+1)}{(N-1)(N-2)(N-3)}\sum_{i=1}^{N}\left(\frac{x_i - \overline{x}}{s_x}\right)^4 - \frac{3(N-1)^2}{(N-2)(N-3)} \quad . \tag{76}$$

Необходимо заметить, что для малых выборок к оценкам асимметрии и эксцесса следует относиться с осторожностью.

Пример 1. Моделирование равномерного и нормального распределений.

В настоящем примере мы построим гистограммы указанных распределений, а также вычислим оценки их параметров и сравним с теоретическими значениями.

Ниже приведен код программы для моделирования случайной величины с равномерным распределением в интервале от 0 до 1.

```
import numpy as np
import matplotlib.pyplot as plt
# 1. Создаем выборку из 200 равномерно распределенных случайных величин
```

```
N = 200
sample = np.random.random(N)
# 2. Находим максимальное и минимальное значение, вычисляем оценки среднего и дисперсии
min_value = np.min(sample)
max_value = np.max(sample)
mean_estimate = np.mean(sample)
variance_estimate = np.var(sample)
# Теоретические оценки для равномерного распределения на [0, 1]
mean_theoretical = 0.5
variance_theoretical = 1/12
print(f"Минимальное значение: {min_value:.4f}")
print(f"Максимальное значение: {max_value:.4f}")
print(f"Оценка среднего: {mean_estimate:.4f} (теоретическое значение: {mean_theoretical:.4f})")
print(f"Оценка дисперсии: {variance_estimate:.4f} (теоретическое значение: {variance_theoretical:.4f})")
# 3. Строим гистограмму распределения и диаграмму накопленных частот
plt.figure(figsize=(12, 6))
# Гистограмма распределения
plt.subplot(1, 2, 1)
plt.hist(sample, bins=20, density=True, edgecolor='black')
plt.xlabel('Значение случайной величины')
plt.ylabel('Плотность вероятности')
plt.title('Гистограмма распределения')
# Теоретическая функция плотности вероятности равномерного распределения
x = np.linspace(0, 1, 100)
pdf_theoretical = np.ones_like(x)
plt.plot(x, pdf_theoretical, 'r-', linewidth=2, label='Теоретическая плотность')
plt.legend()
# Диаграмма накопленных частот
plt.subplot(1, 2, 2)
```

```
cdf_empirical = np.sort(sample)
cdf_empirical = np.arange(len(cdf_empirical)) / (len(cdf_empirical) - 1)
plt.step(np.sort(sample), cdf_empirical, where='post')
plt.xlabel('Значение случайной величины')
plt.ylabel('Вероятность')
plt.title('Диаграмма накопленных частот')
# Теоретическая интегральная функция распределения равномерного распределения
cdf_theoretical = x
plt.plot(x, cdf_theoretical, 'r-', linewidth=2, label='Теоретическая функция распределения')
plt.legend()
plt.tight_layout()
plt.show()
```

В данной программе реализован следующий алгоритм.

1. Создаем выборку из 200 равномерно распределенных случайных величин с помощью функции np.random.random().

2. Находим минимальное и максимальное значение в выборке с помощью функций np.min() и np.max(). Вычисляем оценки среднего и дисперсии с помощью функций np.mean() и np.var(). Сравниваем их с теоретическими значениями для равномерного распределения на [0, 1] по формулам (25),(26).

3. Строим гистограмму распределения с помощью plt.hist() библиотеки matplotlib.pyplot, используя параметр density=True для нормировки гистограммы. Также строим диаграмму накопленных частот с помощью plt.step(). Для сравнения с теоретическими функциями, строим график теоретической плотности вероятности и интегральной функции распределения равномерного распределения на [0, 1].

Вывод результатов:

```
Минимальное значение: 0.0015
Максимальное значение: 0.9903
Оценка среднего: 0.4912 (теоретическое значение: 0.5000)
Оценка дисперсии: 0.0880 (теоретическое значение: 0.0833)
```

Результаты моделирования показывают, что оценки среднего и дисперсии близки к теоретическим значениям, а гистограмма распределения и диаграмма накопленных частот хорошо аппроксимируют теоретические функции равномерного распределения (рис. 10).

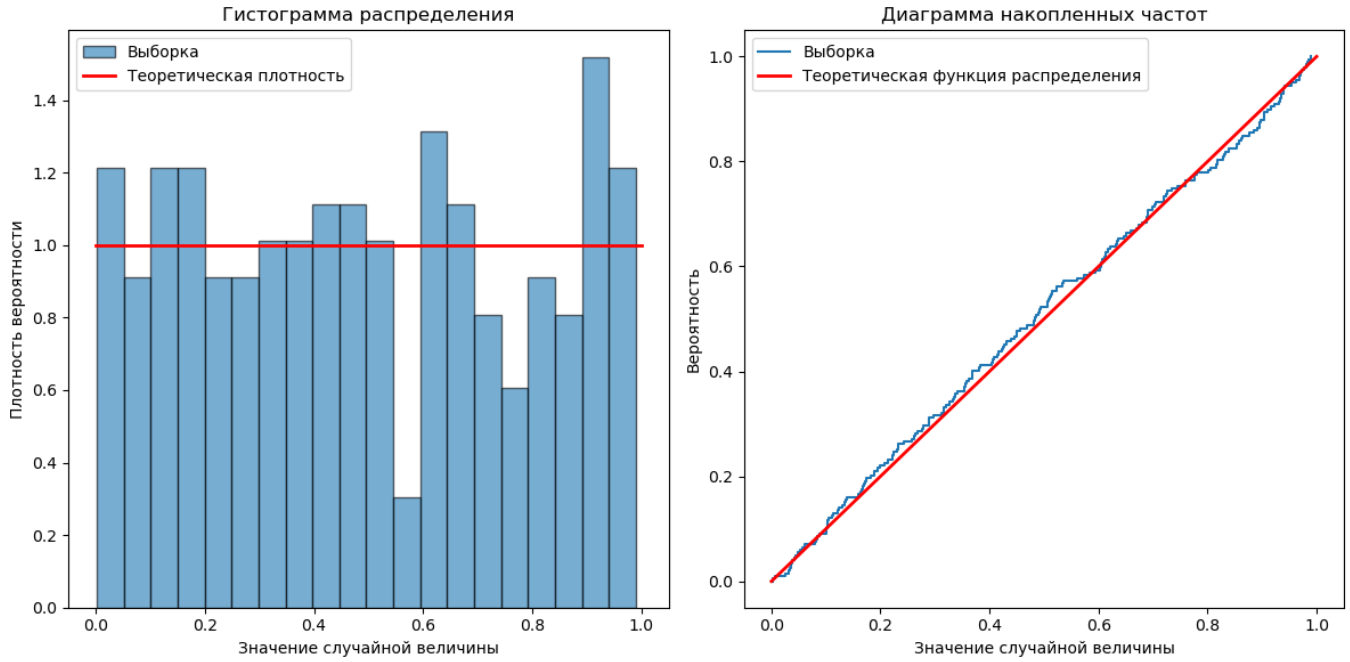

Рисунок 10. Результаты моделирования равномерного распределения

Ниже приведен код программы для моделирования нормального распределения.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
# Задаем параметры нормального распределения
mean = 0        # математическое ожидание
std_dev = 1     # стандартное отклонение
N = 200         # размер выборки
# 1. Создаем выборку из N нормально распределенных случайных величин
sample = np.random.normal(loc=mean, scale=std_dev, size=N)
# 2. Находим максимальное и минимальное значение
max_value = np.max(sample)
```

```
min_value = np.min(sample)
# Вычисляем оценки среднего и дисперсии
sample_mean = np.mean(sample)
sample_variance = np.var(sample)
# Теоретические значения
theoretical_mean = mean
theoretical_variance = std_dev ** 2
# Сравниваем оценки
print(f"Максимальное значение: {max_value}")
print(f"Минимальное значение: {min_value}")
print(f"Оценка среднего: {sample_mean}, Теоретическое среднее: {theoretical_mean}")
print(f"Оценка дисперсии: {sample_variance}, Теоретическая дисперсия: {theoretical_variance}")
# 3. Строим гистограмму распределения
plt.figure(figsize=(12, 6))
# Гистограмма
plt.subplot(1, 2, 1)
plt.hist(sample, bins=20, density=True, alpha=0.6, edgecolor='black', label='Выборка')
plt.title('Гистограмма распределения')
plt.xlabel('Значение')
plt.ylabel('Плотность вероятности')
# Теоретическая функция плотности
x = np.linspace(mean - 4*std_dev, mean + 4*std_dev, 1000)
plt.plot(x, (1/(std_dev * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mean) / std_dev) ** 2), color='red', label='Теоретическая PDF')
plt.legend()
# Диаграмма накопленных частот
plt.subplot(1, 2, 2)
# Сортируем выборку для построения CDF
sorted_sample = np.sort(sample)
cdf = np.arange(1, N + 1) / N
plt.step(sorted_sample, cdf, where='post', label='Выборка')
```

```
plt.title('Диаграмма накопленных частот')
plt.xlabel('Значение')
plt.ylabel('Накопленная вероятность')
# Теоретическая интегральная функция распределения для нормального распределения
cdf_theoretical = stats.norm.cdf(x, loc=mean, scale=std_dev)
plt.plot(x, cdf_theoretical, 'r-', linewidth=2, label='Теоретическая функция распределения')
plt.legend()
plt.tight_layout()
plt.show()
```

В данной программе для моделирования нормального распределения используется функция np.random.normal(loc=mean, scale=std_dev, size=N).

Результат выполнения программы:

Максимальное значение: 2.1626078630930463
Минимальное значение: -2.164693519310872
Оценка среднего: 0.05386987858818422, Теоретическое среднее: 0
Оценка дисперсии: 0.8743875255400695, Теоретическая дисперсия: 1

Оценки математического ожидания и дисперсии несколько отличаются от теоретических значений, тем не менее гистограмма распределения и график диаграммы накопленных частот достаточно близко проходят относительно теоретических функций распределения (рис. 11) .

Аналогичным образом можно моделировать выборки для других видов распределений, реализованных в функциях библиотеки scipy.stats.

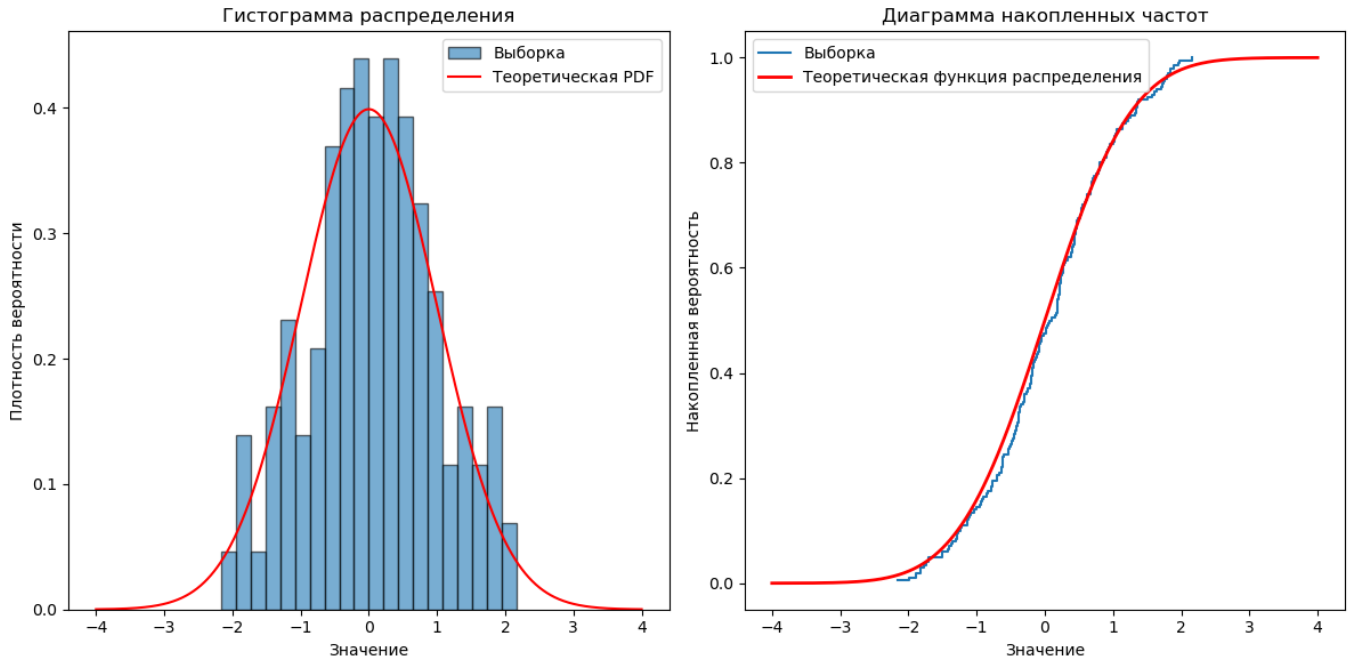

Рисунок 11. Результаты моделирования нормального распределения

Рассмотренные выше оценки параметров случайных величин (72)-(76) называются точечными оценками. Они сами по себе являются случайными величинами, поскольку вычисляются на основании выборки ограниченного объема. Это хорошо видно из рассмотренного примера. При повторном запуске программ оценки математического ожидания и дисперсии для каждого распределения изменяются случайным образом, и хотя остаются относительно близкими к теоретическим значениям, но не совпадают с ними. С увеличением объема выборки значения рассмотренных оценок будут стремиться к истинным значениям параметров. Однако при заданном объеме выборки значение точечной оценки ничего не говорит о степени близости ее к истинному значению параметра. Поэтому более информативный способ оценивания неизвестных параметров состоит не в определении единичного точечного значения, а в построении интервала, в котором с заданной степенью достоверности окажется оцениваемый параметр, т. е. в построении так называемой интервальной оценки параметра.

Интервальной оценкой параметра Θ называется интервал, границы которого $\hat{l}_1$ и $\hat{l}_2$ являются функциями выборочных значений $x_1, x_2 .. x_N$ и который с заданной вероятностью *p* накрывает оцениваемый параметр Θ .

$$P\left\{\hat{l}_1 < \Theta < \hat{l}_2\right\} = p \quad . \tag{77}$$

Интервал $(\hat{l}_1, \hat{l}_2)$ называется *доверительным,* его границы $\hat{l}_1$ и $\hat{l}_2$, являющиеся случайными величинами, — соответственно нижним и верхним *доверительными пределами,* вероятность *p — доверительной вероятностью,* а величина *q=1-p — уровнем значимости,* используемым при построении доверительного интервала. Любая интервальная оценка может быть охарактеризована совокупностью двух чисел: шириной доверительного интервала $L = \hat{l}_2 - \hat{l}_1$, являющейся мерой точности оценивания параметра Θ , и доверительной вероятностью *p,* характеризующей степень достоверности (надежности) результатов. Практически чаще всего используется значение *p*=0,95, несколько реже *p*=0,9 и *p*=0,99 и совсем редко *p*=0,8 и *p*= 0,999. Чем выше значение доверительной вероятности, тем шире будет соответствующий ей доверительный интервал.

Следует заметить, что если точечные оценки математического ожидания и дисперсии, вычисляемые по формулам (72), (73) являются состоятельными для любого закона распределения, то для интервальных оценок способ построения доверительного интервала зависит от вида функции распределения рассматриваемой случайной величины. В дальнейшем мы ограничимся рассмотрением интервальных оценок для случайных величин, распределенных по нормальному закону, поскольку именно он имеет наибольшее практическое значение.

Для выборок небольшого объема интервальные оценки (доверительные интервалы) для рассмотренных выше параметров - математического ожидания и дисперсии имеют следующий вид.

Интервальная оценка математического ожидания

$$\bar{x} - \frac{s_x}{\sqrt{N}} t_{1-q/2} \le m_x \le \bar{x} + \frac{s_x}{\sqrt{N}} t_{1-q/2} \quad , \tag{78}$$

где $\bar{x}$ - точечная оценка математического ожидания (выборочное среднее), $\frac{s_x}{\sqrt{N}}$ - отношение среднеквадратического отклонения выборки к корню из ее объема, называемое также *стандартной ошибкой*, $t_{1-q/2}$ - квантиль $1-\frac{q}{2}$ распределения Стьюдента для выбранной доверительной вероятности *p* при данном объеме выборки *N*.

Распределение Стьюдента применяется для построения доверительных интервалов математического ожидания при малых объемах выборок. Оно более адекватно описывает ситуацию, возникающую при замене генеральной дисперсии распределения σ^2 выборочной дисперсией s^2 . Для выборок большого объема вместо квантиля распределения Стьюдента можно использовать квантиль нормального распределения. Например, следующий код иллюстрирует разницу между квантилями распределения Стьюдента и нормального распределения для разных объемов выборки.

```
from scipy.stats import norm, t
def print_tn(q,df):
    print(f"Доверительная вероятность {q}")
    print(f"Число степеней свободы: {df}")
    print(f"Квантиль распределения Стьюдента:   {t.ppf(q, df)}")
    print(f"Квантиль нормального распределения: {norm.ppf(q, loc=0, scale=1)}")
print_tn(0.95,4)
print_tn(0.95,200)
```

Вывод результата:

```
Доверительная вероятность 0.95
Число степеней свободы: 4
Квантиль распределения Стьюдента:   2.13184678133629
```

Квантиль нормального распределения: 1.6448536269514722
Доверительная вероятность 0.95
Число степеней свободы: 200
Квантиль распределения Стьюдента: 1.652508100910269
Квантиль нормального распределения: 1.6448536269514722

В тех случаях, когда заведомо известно, что оценка математического ожидания может быть только ниже или только выше истинного значения, необходимо вычислять односторонний доверительный интервал (соответственно левая или правая части неравенства (78)). В одностороннем интервале значение уровня значимости при вычислении критерия Стьюдента следует принимать равным 1-*q*, а не 1-*q*/2, как в формуле (78).

Интервальная оценка дисперсии

$$\frac{(N-1)s_x^2}{\chi^2_{1-q/2}} \le \sigma_x^2 \le \frac{(N-1)s_x^2}{\chi^2_{q/2}} \quad , \qquad (79)$$

где s_x^2 - точечная оценка дисперсии, $\chi^2_{1-q/2}$ и $\chi^2_{q/2}$ - квантили $1-\frac{q}{2}$ и

распределения χ^2 для выбранной доверительной вероятности *p* и числа степеней свободы $df = N-1$. Квантили распределения χ^2 могут быть вычислены при помощи функции scipy.stats.chi2.ppf(p, df).

При вычислении оценки дисперсии возникают некоторые особенности, связанные с выбором числа степеней свободы (Degrees of Freedom, dof). В знаменателе формулы (73) стоит выражение $N-1$ - число степеней свободы , где N - объем выборки. Такая оценка называется несмещенной оценкой дисперсии. Уменьшение числа степеней на единицу по сравнению с N вызвано тем, что при вычислении дисперсии в формуле используется оценка математического ожидания по выборке (среднее значение), то есть на выборку накладывается одна связь. Если бы было известно теоретическое значение m_x , его следовало бы подставить в формулу (73) вместо среднего по выборке $\bar{x}$, а в знаменатель формулы поставить N . Такая оценка дисперсии

называется оценкой дисперсии генеральной совокупности. На практике m_x обычно не известно и следует пользоваться несмещенной оценкой (73). Эта оценка дисперсии также подразумевается и в формуле (79), поэтому в числителе здесь стоит $N-1$. При больших N разница между двумя оценками становится незначительной, но при малых N , она заметна. Число связей, накладываемых на выборку при вычислении дисперсии может быть и больше 1. Такая ситуация возникает в дисперсионном анализе. Поэтому в общем случае в знаменателе формулы для вычисления оценки дисперсии следует записать $N-ddof$, где $ddof$ - число связей, накладываемых на выборку (Delta Degrees of Freedom). В функции np.var библиотеки numpy используется именно этот вариант, причем по умолчанию установлено $ddof=0$, то есть по умолчанию вычисляется смещенная оценка дисперсии. Мы однако не учитывали это обстоятельство в коде программ Примера 1, поскольку с одной стороны, эти тонкости не были объяснены, а с другой стороны, объем выборки при оценке функций распределения достаточно большой и разница между смещенной и несмещенной оценками незначительна. В дальнейшем, однако, мы будем учитывать вышеописанные особенности и в параметрах функции np.var будем указывать конкретное значение $ddof$.

Пример 2. Вычисление точечных и интервальных оценок математического ожидания и дисперсии нормально распределенной случайной величины.

Ниже приведен пример кода для моделирования выборки нормально распределенной случайной величины и вычисления точечных и интервальных оценок математического ожидания и дисперсии.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
# Задаем параметры нормального распределения
mu = 10  # математическое ожидание
sigma = 2  # стандартное отклонение
N = 100  # объем выборки
# Генерируем выборку
sample = np.random.normal(mu, sigma, N)
# Вычисляем точечные оценки
```

```
sample_mean = np.mean(sample)  # точечная оценка математического ожидания
sample_variance = np.var(sample, ddof=1)  # точечная оценка дисперсии
# Вычисляем доверительные интервалы для математического ожидания
confidence_level = 0.95
alpha = 1 - confidence_level
t_critical = stats.t.ppf(1 - alpha/2, df=N-1)  # критическое значение t-распределения
# Доверительный интервал для математического ожидания
margin_of_error = t_critical * (np.std(sample, ddof=1) / np.sqrt(N))
confidence_interval_mean = (sample_mean - margin_of_error, sample_mean + margin_of_error)
# Вычисляем доверительный интервал для дисперсии
chi2_lower = stats.chi2.ppf(alpha/2, df=N-1)
chi2_upper = stats.chi2.ppf(1 - alpha/2, df=N-1)
confidence_interval_variance = ((N-1) * sample_variance / chi2_upper, (N-1) * sample_variance / chi2_lower)
# Вывод результатов
print(f"Точечная оценка математического ожидания: {sample_mean:.2f}")
print(f"Доверительный интервал для математического ожидания: {confidence_interval_mean}")
print(f"Точечная оценка дисперсии: {sample_variance:.2f}")
print(f"Доверительный интервал для дисперсии: {confidence_interval_variance}")
```

Вывод результатов расчета:

```
Точечная оценка математического ожидания: 10.18
Доверительный интервал для математического ожидания: (9.800729838032675, 10.553423801200962)
Точечная оценка дисперсии: 3.60
Доверительный интервал для дисперсии: (2.7732841343298076, 4.854763079774836)
```

В данной программе генерируется выборка sample объемом N = 100 нормально распределенных случайных чисел с заданными значениями математического ожидания mu=10 и стандартного отклонения sigma = 2 с помощью функции np.random.normal(mu, sigma, N). Для данной выборки вычисляются точечные оценки математического ожидания и дисперсии с

помощью функций np.mean(sample) и np.var(sample, ddof=1), соответственно. Параметр ddof=1 указывает, что для дисперсии вычисляется несмещенная оценка. При вычислении интервальной оценки математического ожидания по формуле (78) используется функция stats.t.ppf(1 - alpha/2, df=N-1) для квантиля распределения Стьюдента. Интервальная оценка дисперсии вычисляется по формуле (79) с использованием функций stats.chi2.ppf(1 - alpha/2, df=N-1) и stats.chi2.ppf(alpha/2, df=N-1) для квантилей распределения хи-квадрат в левой и правой границах доверительного интервала, соответственно. Как видно из результатов расчета, точечные оценки находятся внутри соответствующих доверительных интервалов, внутри которых находятся и теоретические значения математического ожидания и дисперсии, использованные при генерации выборки.

При меньшем значении доверительной вероятности ширина доверительных интервалов будет уменьшаться, однако при этом можно наблюдать ситуации, когда истинные значения параметров будут не попадать в границы соответствующего доверительного интервала. Эта ситуация связана с так называемой ошибкой первого рода, возникающей при проверке статистических гипотез. С увеличением доверительной вероятности доверительные интервалы расширяются, и вероятность ошибки первого рода будет меньшей.

Пример 3. Определение объема выборки, необходимого для оценки математического ожидания с заданной точностью.

На практике часто возникает задача определения необходимого числа опытов для получения значения параметра m_x с заданной точностью. Строго говоря, эту задачу можно решить только в том случае, если известно истинное значение дисперсии генеральной совокупности. Пусть известное среднеквадратическое отклонение генеральной совокупности равно σ и необходимо определить потребное количество опытов N для обеспечения заданной величины ошибки δ измерения исследуемого параметра. Для этого необходимо при заданной величине доверительной вероятности выразить N из формулы для интервальной оценки параметра

$$\frac{\sigma}{\sqrt{N}} t_{1-\frac{q}{2}} = \delta \quad . \tag{80}$$

Сделать это не совсем просто, поскольку квантиль распределения Стьюдента $t_{1-\frac{q}{2}}$ сам является функцией от N, и уравнение (80) относительно N можно решить только численным методом. Вместе с тем, выше было показано, что при большом числе опытов распределение Стьюдента приближается к нормальному. В этом случае квантиль распределения Стьюдента в формуле можно заменить на квантиль нормального распределения $z_{1-\frac{q}{2}}$, не зависящий от числа опытов. Тогда потребное количество опытов N для заданной точности δ можно рассчитать, выразив его из формулы

$$\frac{\sigma}{\sqrt{N}} z_{1-\frac{q}{2}} = \delta \quad . \tag{81}$$

В результате получим

$$N = \left(\frac{\sigma z_{1-q/2}}{\delta} \right)^2 \quad . \tag{82}$$

Ниже приведен код программы, решающей данную задачу.

```
import math
from scipy.stats import norm
def calculate_experiments(std_dev, precision, confidence_level):
    z_score = abs(norm.ppf((1 - confidence_level) / 2))
    n = (z_score * std_dev / precision) ** 2
    return math.ceil(n)
# Пример использования
true_std_dev = 1
desired_precision = 0.1
confidence = 0.95
num_experiments = calculate_experiments( true_std_dev, desired_precision, confidence)
print(f"Необходимое число опытов: {num_experiments}")
```

Результат вычисления:

Необходимое число опытов: 385

В данной программе вычисление квантиля нормального распределения осуществляется с помощью функции norm.ppf библиотеки scipy.stats.

3.1.2. Многомерные случайные величины

Анализ двумерных случайных величин чаще всего сводится к оценке статистической связи между этими величинами, характеризуемой величиной ковариации или коэффициентом корреляции Пирсона. Такой анализ соответственно называется ковариационным или корреляционным анализом. Соответствующие формулы для точечных оценок ковариации и коэффициента корреляции имеют вид

$$\hat{\mathrm{cov}}(x,y)=\frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})(y_i-\bar{y}) \quad , \tag{83}$$

$$\hat{\rho}_{xy}=\frac{\hat{\mathrm{cov}}(x,y)}{s_x s_y} \quad . \tag{84}$$

Для выборки большого объема интервальная оценка коэффициента корреляции имеет вид

$$\hat{\rho}_{xy}-\frac{u_p(1-\hat{\rho}_{xy}^2)}{\sqrt{N}}\le\rho_{xy}\le\hat{\rho}_{xy}+\frac{u_p(1-\hat{\rho}_{xy}^2)}{\sqrt{N}} \quad , \tag{85}$$

где $\hat{\rho}_{xy}$ - точечная оценка коэффициента корреляции, u_p - квантиль нормального распределения для выбранной доверительной вероятности *p*.

При малом числе экспериментов и сравнительно высокой корреляции распределение коэффициента корреляции существенно отличается от нормального. В этом случае для построения доверительного интервала можно воспользоваться преобразованием Фишера

$$\hat{\rho}_{xy} = thz = \frac{e^{2z} - 1}{e^{2z} + 1} \quad , \tag{86}$$

отсюда

$$z = \frac{1}{2} \ln \frac{1 + \hat{\rho}_{xy}}{1 - \hat{\rho}_{xy}} \quad . \tag{87}$$

Доверительный интервал для *z* вычисляется по формуле

$$z - \frac{u_p}{\sqrt{N-3}} \le m_z \le z + \frac{u_p}{\sqrt{N-3}} \quad . \tag{88}$$

После нахождения левой и правой границ доверительного интервала для *z* можно по формуле (86) вычислить соответствующие границы доверительного интервала для ρ_{xy} .

Для вычисления ковариации между двумя выборками *x* и *y* используется функция np.cov() библиотеки numpy:

```
cov_matrix = np.cov(x, y)
```

Результатом данной команды будет ковариационная матрица размера 2x2, где диагональные элементы - дисперсии выборок *x* и *y* , а недиагональные - ковариации.

Для вычисления коэффициента корреляции Пирсона между *x* и *y* используется функция scipy.stats.pearsonr():

```
rho, p = scipy.stats.pearsonr(x, y)
```

Функция возвращает коэффициент корреляции rho и p-value p. Коэффициент rho лежит в диапазоне [-1, 1], где -1 означает полную отрицательную линейную зависимость, 0 - отсутствие линейной зависимости, а 1 - полную положительную линейную зависимость. P-value (вероятностное значение) - представляет собой вероятность того, что наблюдаемое значение коэффициента корреляции Пирсона (rho) могло бы возникнуть случайно, если на самом деле

между переменными нет линейной связи, то есть при условии, что верна нулевая гипотеза. Нулевая гипотеза при вычислении коэффициента корреляции заключается в том, что истинный коэффициент корреляции между двумя переменными равен нулю. Это обозначается как $H_0: \rho = 0$. В противовес нулевой гипотезе формулируется альтернативная гипотеза $H_1: \rho \neq 0$, которая утверждает, что существует значимая линейная зависимость между переменными. При проверке нулевой гипотезы о коэффициенте корреляции значение p-value, которое отвечает принятию нулевой гипотезы, обычно составляет 0.05 или больше. Это означает, что если p-value больше или равно 0.05, то нет достаточных оснований для отклонения нулевой гипотезы, которая утверждает, что истинный коэффициент корреляции равен нулю. Более подробно статистические гипотезы рассмотрены в следующем разделе.

P-value рассчитывается на основе t-распределения, где t-статистика вычисляется из коэффициента корреляции и размера выборки. Формула для t-статистики выглядит следующим образом:

$$t = \frac{\rho \sqrt{N-2}}{\sqrt{1-\rho^2}} \tag{89}$$

где ρ — коэффициент корреляции, а N — размер выборки. Затем p-value определяется как двустороннее значение для t-распределения с n-2 степенями свободы.

Кроме коэффициента корреляции Пирсона также можно вычислить ранговую корреляцию Спирмена с помощью функции scipy.stats.spearmanr()

```
rho, p = scipy.stats.spearmanr(x, y)
```

Эта функция устойчива к выбросам и подходит для оценки нелинейных зависимостей. Она используется для оценки корреляции между двумя ранговыми переменными или когда данные не соответствуют нормальному распределению. Метод вычисления коэффициента корреляции Спирмена основан на ранжировании данных. Каждое значение заменяется его рангом, и затем применяется формула для вычисления корреляции. Это делает его более устойчивым к выбросам. Диапазон значений коэффициента корреляции Спирмена, как и коэффициент корреляции Пирсона, варьируется от -1 до 1, но интерпретируется как степень монотонной зависимости, что может включать нелинейные связи.

Пример 4. Вычисление оценок коэффициента корреляции.

Исследуем статистическую связь между пятью случайными величинами *A, B, C, D, E*, которые зададим следующим образом. Пусть ξ - случайная величина, имеющая стандартное нормальное распределение. Определим случайные величины $A=\xi$, $B=A+\xi$, $C=B+\xi$, $D=C+\xi$, $E=D+\xi$. То есть в ряду *A, B, C, D, E* каждая последующая величина представляет собой сумму предыдущей и стандартной нормально распределенной случайной величины ξ. Создадим выборку указанных случайных величин и вычислим точечные и интервальные оценки их парных коэффициентов корреляции.

Ниже приведен код программы для решения данной задачи.

```
import numpy as np
from scipy.stats import pearsonr
from scipy.stats import norm
# Задаем размер выборки
n = 1000
# Генерируем выборку стандартной нормально распределенной случайной величины x
def x_rnd():
    return np.random.normal(loc=0, scale=1, size=n)
# Вычисляем случайные величины A, B, C, D, E
A = x_rnd()
B = A + x_rnd()
C = B + x_rnd()
D = C + x_rnd()
E = D + x_rnd()
# Вычисляем точечные оценки парных коэффициентов корреляции
r_AB, p_AB = pearsonr(A, B)
r_AC, p_AC = pearsonr(A, C)
r_AD, p_AD = pearsonr(A, D)
r_AE, p_AE = pearsonr(A, E)
r_BC, p_BC = pearsonr(B, C)
```

```
r_BD, p_BD = pearsonr(B, D)
r_BE, p_BE = pearsonr(B, E)
r_CD, p_CD = pearsonr(C, D)
r_CE, p_CE = pearsonr(C, E)
r_DE, p_DE = pearsonr(D, E)
# Выводим точечные оценки
print("Точечные оценки парных коэффициентов корреляции:")
print(f"r(A, B) = {r_AB:.4f}")
print(f"r(A, C) = {r_AC:.4f}")
print(f"r(A, D) = {r_AD:.4f}")
print(f"r(A, E) = {r_AE:.4f}")
print(f"r(B, C) = {r_BC:.4f}")
print(f"r(B, D) = {r_BD:.4f}")
print(f"r(B, E) = {r_BE:.4f}")
print(f"r(C, D) = {r_CD:.4f}")
print(f"r(C, E) = {r_CE:.4f}")
print(f"r(D, E) = {r_DE:.4f}")
# Вычисляем интервальные оценки с доверительной вероятностью 95%
alpha = 0.05
z = norm.ppf(1 - alpha/2)
se = 1 / np.sqrt(n - 3)
print("\nИнтервальные оценки парных коэффициентов корреляции с доверительной вероятностью 95%:")
print(f"r(A, B) = {r_AB:.4f} ± {z * se:.4f}")
print(f"r(A, C) = {r_AC:.4f} ± {z * se:.4f}")
print(f"r(A, D) = {r_AD:.4f} ± {z * se:.4f}")
print(f"r(A, E) = {r_AE:.4f} ± {z * se:.4f}")
print(f"r(B, C) = {r_BC:.4f} ± {z * se:.4f}")
print(f"r(B, D) = {r_BD:.4f} ± {z * se:.4f}")
print(f"r(B, E) = {r_BE:.4f} ± {z * se:.4f}")
print(f"r(C, D) = {r_CD:.4f} ± {z * se:.4f}")
```

```
print(f"r(C, E) = {r_CE:.4f} ± {z * se:.4f}")
print(f"r(D, E) = {r_DE:.4f} ± {z * se:.4f}")
```

Результаты расчета:

Точечные оценки парных коэффициентов корреляции:

r(A, B) = 0.7094

r(A, C) = 0.5557

r(A, D) = 0.4847

r(A, E) = 0.4394

r(B, C) = 0.8166

r(B, D) = 0.7128

r(B, E) = 0.6275

r(C, D) = 0.8703

r(C, E) = 0.7624

r(D, E) = 0.8873

Интервальные оценки парных коэффициентов корреляции с доверительной вероятностью 95%:

r(A, B) = 0.7094 ± 0.0621

r(A, C) = 0.5557 ± 0.0621

r(A, D) = 0.4847 ± 0.0621

r(A, E) = 0.4394 ± 0.0621

r(B, C) = 0.8166 ± 0.0621

r(B, D) = 0.7128 ± 0.0621

r(B, E) = 0.6275 ± 0.0621

r(C, D) = 0.8703 ± 0.0621

r(C, E) = 0.7624 ± 0.0621

r(D, E) = 0.8873 ± 0.0621

Как можно видеть, в последовательности рассматриваемых случайных величин *A*, *B*, *C*, *D*, *E* по мере удаления от *A* коэффициент корреляции между предыдущей в ряду и последующей случайной величиной возрастает, тогда как с увеличением расстояния между ними он, наоборот, уменьшается. То есть по мере удаления друг от друга в рассматриваемом ряду случайные величины становятся зависимыми в меньшей мере.

Интервальные оценки показывают возможный разброс коэффициентов корреляции. Их изменение в рассматриваемой последовательности случайных величин аналогично изменению точечных оценок.

В рассмотренном примере вычисленные оценки коэффициентов корреляции во всех случаях оказались достаточно высокими, что отражает наличие статистической зависимости между исследуемыми случайными величинами. На практике может возникнуть ситуация, когда доверительный интервал для коэффициента корреляции включает значение 0. В этом случае можно сделать вывод, что такие случайные величины являются независимыми.

Множественный корреляционный анализ. Рассмотрим проблему оценки статистической связи $p+1$ переменных $Y, X_1, \ldots, X_p$. Предполагается, что эти величины имеют совместное многомерное нормальное распределение. Пусть это распределение имеет средние $\mu_y, \mu_1, \ldots, \mu_p$ и дисперсии $\sigma_y^2, \sigma_1^2, \ldots, \sigma_p^2$ соответственно. Обозначим ковариацию Y с X_i через σ_{yi} и ковариацию X_i с X_j через σ_{ij} для $i, j = 1, \ldots, p$. Определим далее коэффициенты корреляции $\rho_i = \sigma_{yi} / (\sigma_y \sigma_i)$ и $\rho_{x_i x_j} = \sigma_{ij} / (\sigma_i \sigma_j)$.

Для данных значений $X_1 = x_1, \ldots, X_p = x_p$ существует подмножество соответствующих значений Y . Их распределение, называемое *условным распределением* Y *при данных* $X_1 = x_1, \ldots, X_p = x_p$, является нормальным со средним значением

$$\mu_{y \cdot x_1 \ldots x_p} = \mu_y + \beta_1 (x_1 - \mu_1) + \ldots + \beta_p (x_p - \mu_p) \quad , \qquad (90)$$

которое называется *условным ожиданием* Y при данных $X_1 = x_1, \ldots, X_p = x_p$ или *регрессией* Y по $X_1, \ldots, X_p$. Величины $\beta_1, \ldots, \beta_p$ называются *(частными) коэффициентами регрессии* и являются функциями дисперсий и ковариаций. Дисперсия этого условного распределения дается величиной

$$\sigma^2=\sigma_y^2\left(1-\rho_{y\cdot x_1\ldots x_p}^2\right) \quad , \qquad (91)$$

где $\rho_{y\cdot x_1\ldots x_p}$ - положительный квадратный корень из $\rho_{y\cdot x_1\ldots x_p}^2$ называется множественным коэффициентом корреляции между Y и $X_1,\ldots,X_p$.

Если ввести случайную величину $e=Y-\mu_{y\cdot x_1\ldots x_p}$, то условное распределение e при данных $X_1=x_1,\ldots,X_p=x_p$ будет $N\left(0,\sigma^2\right)$. Используя условное распределение, можно написать

$$Y=\beta_0+\beta_1 x_1+\ldots+\beta_p x_p+e \quad , \qquad (92)$$

где $\beta_0=\mu_y-\beta_1\mu_1-\ldots-\beta_p\mu_p$ и e распределено по $N\left(0,\sigma^2\right)$.

Множественный коэффициент корреляции $\rho_{y\cdot x_1\ldots x_p}$ является мерой линейной зависимости между Y и набором переменных $\left\{X_1,\ldots,X_p\right\}$, причем $0\le\rho_{y\cdot x_1\ldots x_p}\le 1$. Нулевое значение этого коэффициента указывает, что Y не зависит (линейно) от набора переменных $\left\{X_1,\ldots,X_p\right\}$, а значение 1 указывает полную линейную зависимость, при которой переменная Y равна линейной комбинации переменных $X_1,\ldots,X_p$.

Разрешая уравнение (91) относительно множественного коэффициента корреляции, получаем $\rho_{y\cdot x_1\ldots x_p}^2=(\sigma_y^2-\sigma^2)/\sigma_y^2$. То есть квадрат множественного коэффициента корреляции равен доле дисперсии Y , «объясненной» регрессионной зависимостью с $X_1,\ldots,X_p$. В свою очередь, $\left(1-\rho_{y\cdot x_1\ldots x_p}^2\right)^{1/2}$ есть доля стандартного отклонения Y , оставшаяся «не

объясненной» зависимостью от $X_1,\ldots,X_p$. Например, если множественный коэффициент корреляции равен 0,9, остается 44 % необъясненного стандартного отклонения Y .

Множественный коэффициент корреляции инвариантен относительно невырожденных линейных преобразований переменных. В частности, он инвариантен к изменению масштаба или начала отсчета шкалы измерения $Y,X_1,\ldots,X_p$.

Помимо множественного коэффициента корреляции на практике находят применение также *частные коэффициенты корреляции*. Частный коэффициент корреляции используется как мера линейной зависимости между двумя какими-либо переменными из $Y,X_1,\ldots,X_p$ после вычитания «эффекта», обусловленного взаимодействием этих двух переменных с некоторым непустым подмножеством из оставшихся $p-1$ переменных. В частности, таким способом можно измерять зависимость между Y и независимой переменной X_m после учета линейной зависимости Y от некоторого подмножества k переменных, содержащегося среди $p-1$ независимых переменных X_i , $i=1,\ldots,p$, $i\neq m$. Теория частного коэффициента корреляции основана на изучении двух условных распределений.

Пусть l и h - две какие-либо переменные из набора $Y,X_1,\ldots,X_p$ и c - некоторое непустое подмножество из оставшихся $p-1$ переменных. Определим величины $Z_1=l-\mu_{l\cdot c}$ и $Z_2=h-\mu_{h\cdot c}$. Здесь $\mu_{l\cdot c}$ и $\mu_{h\cdot c}$ - соответственно условные ожидаемые значения l и h при данном c . *Частный коэффициент корреляции между* l *и* h *при фиксированных значениях переменных из* c *есть*

$$\rho_{lh\cdot c}=\rho_{Z_1Z_2} \quad , \tag{93}$$

где $\rho_{Z_1Z_2}$ - простой коэффициент корреляции между Z_1 и Z_2 .

Если $l=Y$, $h=X_m$, $m=1,\ldots,p$, а c составляют все оставшиеся $p-1$ независимые переменные, то такой коэффициент корреляции называется частным коэффициентом корреляции первого порядка и будет обозначаться через $\rho_{yx_m\cdot c}$. Если $l=Y$, $h=X_m$, а c составляет подмножество из первых k независимых переменных $\{X_1,\ldots X_k\}$, где $1\le k<m\le p$, то соответствующий частный коэффициент корреляции называется коэффициентом *k-го порядка* и будет обозначаться через $\rho_{yx_m\cdot x_1\ldots x_k}$.

Частный коэффициент корреляции $\rho_{lh\cdot c}$ есть мера линейной зависимости между l и h , когда величины переменных из c фиксированы. Значения этого коэффициента корреляции заключены между –1 и +1; значение нуль указывает на то, что l и h независимы, когда величины переменных из c фиксированы.

Имеет место следующее тождество между множественным и частным коэффициентами корреляции для набора переменных $Y,X_1,\ldots,X_{k-1},X_k$, $k=2,\ldots,p$:

$$1-\rho^2_{y\cdot x_1\ldots x_k}=\left(1-\rho^2_{y\cdot x_1\ldots x_{k-1}}\right)\left(1-\rho^2_{yx_k\cdot x_1\ldots x_{k-1}}\right) \quad . \tag{94}$$

Это тождество следует из того, что

$$\sigma^2\left(Y\,|\,X_1,\ldots,X_k\right)=\sigma^2\left(Y\,|\,X_1,\ldots,X_{k-1}\right)\left(1-\rho^2_{yx_k\cdot x_1\ldots x_{k-1}}\right) \quad , \tag{95}$$

где $\sigma^2\left(Y\,|\,X_1,\ldots,X_i\right)$ - условная дисперсия Y при заданных значениях $X_1,\ldots,X_i$, $i=1,\ldots,p$. Так как

$$\rho^2_{yx_k\cdot x_1\ldots x_{k-1}}=\frac{\sigma^2\left(Y\,|\,X_1,\ldots,X_{k-1}\right)-\sigma^2\left(Y\,|\,X_1,\ldots,X_k\right)}{\sigma^2\left(Y\,|\,X_1,\ldots,X_{k-1}\right)} \quad ,$$

то квадрат частного коэффициента корреляции можно определить как долю остаточной дисперсии Y, «объясненной» добавлением переменной X_k к набору $\left\{X_1,\ldots,X_{k-1}\right\}$.

Верно соотношение

$$\rho_{yx_m\cdot c}=\beta_m\left[\frac{\sigma^2\left(X_m|c\right)}{\sigma^2\left(Y|c\right)}\right]^{1/2}\quad,\quad m=1,\ldots,p\quad,$$

где c состоит из всех оставшихся $p-1$ переменных, а $\sigma^2\left(X_m|c\right)$ - условная дисперсия X_m при фиксированных значениях переменных из c. Поэтому проверка гипотезы $\beta_m=0$ эквивалентна проверке гипотезы $\rho_{yx_m\cdot c}=0$.

Частные коэффициенты корреляции могут быть вычислены на основе рекуррентных соотношений следующим образом. Если l,h,d - три различные переменные из множества $\left\{Y,X_1,\ldots,X_p\right\}$, то все частные коэффициенты корреляции первого порядка даются соотношением

$$\rho_{lh\cdot d}=\frac{\rho_{lh}-\rho_{ld}\,\rho_{hd}}{\sqrt{\left(1-\rho_{ld}^2\right)\left(1-\rho_{hd}^2\right)}}\quad, \tag{96}$$

где все величины в правой части суть простые коэффициенты корреляции. Далее, последовательно применяя формулу

$$\rho_{lh\cdot cd}=\frac{\rho_{lh\cdot c}-\rho_{ld\cdot c}\,\rho_{hd\cdot c}}{\sqrt{\left(1-\rho_{ld\cdot c}^2\right)\left(1-\rho_{hd\cdot c}^2\right)}}\quad, \tag{97}$$

где c - любое подмножество оставшихся переменных, можно получить частные коэффициенты корреляции любого порядка.

Рассмотрим, как выглядят формулы множественного корреляционного анализа в матричных обозначениях.

Пусть $Z=(Y,X_1,\ldots,X_p)^T$ - вектор случайных переменных размерности $p+1$. Предположим, что этот вектор имеет многомерное нормальное распределение с вектором средних значений $E(Z)=(\mu_y,\mu_1,\ldots,\mu_p)^T$ и матрицей ковариаций размерности $(p+1)\times(p+1)$

$$\Sigma_Z=\begin{bmatrix}\sigma_y^2 & \sigma_{y1} & \sigma_{y2} & \ldots & \sigma_{yp}\\ \sigma_{y1} & \sigma_1^2 & \sigma_{12} & \ldots & \sigma_{1p}\\ \ldots & \ldots & \ldots & \ldots & \ldots\\ \sigma_{yp} & \sigma_{1p} & \sigma_{2p} & \ldots & \sigma_p^2\end{bmatrix}$$

Чтобы получить уравнения для определения коэффициентов регрессии $\beta_1,\ldots\beta_p$ и множественного коэффициента корреляции $\rho_{y\cdot x_1\ldots x_p}$, запишем Z в виде составного вектора $Z=(Y,X^T)^T$, где $X=(X_1,\ldots,X_p)^T$ - вектор размерности p. Вектор средних значений и ковариационная матрица аналогичным образом разлагаются на части, т.е. $E(Z)=(\mu_y,\mu_x^T)^T$, где $\mu_x=(\mu_1,\ldots,\mu_p)^T$ и

$$\Sigma_Z=\begin{pmatrix}\sigma_y^2 & \Sigma_{yx}\\ \Sigma_{xy} & \Sigma_{xx}\end{pmatrix}$$

Подматрицы Σ_{yx}, Σ_{xy} и Σ_{xx} имеют размерности $1\times p$, $p\times 1$ и $p\times p$ соответственно. Заметим, что $\Sigma_{yx}^T=\Sigma_{xy}$. Таким образом, условное распределение Y при заданном значении $X=x$ является нормальным со средним

$$\mu_{y\cdot x_1\ldots x_p}=\mu_y+\Sigma_{xx}^{-1}\Sigma_{xy}(x-\mu_x)$$

и дисперсией

$$\sigma^2=\sigma_y^2-\Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy}$$

С учетом (91) отсюда следует, что

$$\rho_{y\cdot x_1\ldots x_p}=\sqrt{\Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy}}/\sigma_y \quad .$$

Перейдем к частному коэффициенту корреляции $\rho_{lh\cdot c}$. Перенумеруем переменные X_i так, чтобы $c=\left(X_1,\ldots,X_k\right)$. Определим случайные векторы $W_1=\left(l,h\right)^T$ размерности 2 и $W_2=\left(X_1,\ldots,X_k\right)^T$ размерности *k*. Вектор W_1 имеет двумерное нормальное распределение с вектором средних $E\left(W_1\right)=\left(\mu_l,\mu_h\right)^T$ и ковариационной матрицей

$$\Sigma_{w_1w_1}=\begin{bmatrix}\sigma_l^2 & \sigma_{lk}\\ \sigma_{lk} & \sigma_k^2\end{bmatrix}$$

Аналогично распределение W_2 есть *k*-мерное нормальное распределение с вектором средних значений $E\left(W_2\right)=\left(\mu_1,\ldots,\mu_k\right)^T$ и матрицей ковариаций

$$\Sigma_{w_2w_2}=\begin{bmatrix}\sigma_1^2 & \sigma_{12}\ldots\sigma_{1k}\\ \ldots\ldots\ldots\ldots & \\ \sigma_{1k} & \sigma_{2k}\ldots\sigma_k^2\end{bmatrix}$$

Определим теперь $2\times k$ матрицу ковариаций между W_1 и W_2 :

$$\Sigma_{w_1w_2}=\begin{bmatrix}\sigma_{l1} & \sigma_{l2}\ldots\sigma_{lk}\\ \sigma_{k1} & \sigma_{k2}\ldots\sigma_{kk}\end{bmatrix}$$

При этом $\Sigma_{w_1w_2}=\Sigma_{w_2w_1}^T$.

Условное распределение W_1 при фиксированных значениях элементов W_2 , например $W_2=w_2$, будет двумерным нормальным распределением с вектором средних значений

$$E\left(W_1\right)+\Sigma_{w_1w_2}\Sigma_{w_2w_2}^{-1}\left(w_2-E\left(W_2\right)\right) \quad ,$$

который называется *условным математическим ожиданием* W_1 при заданном значении $W_2=w_2$, и ковариационной матрицей

$$\Sigma_{w_1w_2}-\Sigma_{w_1w_2}\Sigma_{w_2w_2}^{-1}\Sigma_{w_2w_1}=\begin{bmatrix}\sigma_{l\cdot c}^2\sigma_{lh\cdot c}\\ \sigma_{lh\cdot c}\sigma_{h\cdot c}^2\end{bmatrix}$$

Теперь частный коэффициент корреляции можно записать в виде

$$\rho_{lh\cdot c}=\sigma_{lh\cdot c}/\left(\sigma_{l\cdot c}\sigma_{h\cdot c}\right) \quad .$$

Перейдем к вопросу оценивания множественного и частного коэффициентов корреляции. Оценку множественного коэффициента корреляции будем обозначать через $r_{y\cdot x_1\ldots x_p}$. Эта оценка может быть получена в ходе линейного множественного регрессионного анализа. Пусть для получения оценок множественного и частного коэффициентов корреляции имеется выборка случайных векторов $z_i=\left(y_i,x_{1i},\ldots,x_{pi}\right)^T$, $i=1,\ldots,n$. Оценкой максимального правдоподобия для $E\left(Z\right)$ будет вектор

$$\bar{z}=\frac{1}{n}\sum_{i=1}^{n}z_i \quad ,$$

а несмещенной оценкой для Σ_z - матрица

$$\mathbf{S}_z=\frac{1}{n-1}\sum_{i=1}\left(\mathbf{z}_i-\bar{\mathbf{z}}\right)\left(\mathbf{z}_i-\bar{\mathbf{z}}\right)^T=\begin{bmatrix}s_y^2 s_{y1}s_{y2}\ldots s_{yp}\\ s_{y1}s_1^2 s_{12}\ldots s_{1p}\\ \ldots\ldots\ldots\ldots\\ s_{yp}s_{1p}s_{2p}\ldots s_p^2\end{bmatrix}$$

Несмещенные оценки Σ_{xx} , Σ_{xy} и Σ_{yx} получаются разбиением матрицы S_z на блоки подобно матрице Σ_z . Обозначим соответствующие оценки через S_{xx}, S_{xy} и S_{yx} .

Тогда выборочный множественный коэффициент корреляции равен

$$r_{y \cdot x_1 \ldots x_p} = \frac{\sqrt{S_{yx} S_{xx}^{-1} S_{xy}}}{s_y^2} \quad . \tag{98}$$

Для получения оценки частного коэффициента корреляции определим матрицы $S_{w_1 w_1}$, $S_{w_1 w_2}$ и $S_{w_2 w_2}$ таким же образом, как и их аналоги по генеральной совокупности, заменяя соответствующие параметры выборочными дисперсиями и ковариациями. Тогда оценка частного коэффициента корреляции имеет вид

$$r_{lh \cdot c} = s_{lh \cdot c} / \left(s_{l \cdot c} s_{h \cdot c} \right) \quad . \tag{99}$$

3.1.3. Статистические гипотезы

Статистическая гипотеза есть некоторое предположение относительно свойств генеральной совокупности, из которой извлекается выборка. *Критерий статистической гипотезы* — это правило, позволяющее принять или отвергнуть данную гипотезу на основании выборки. При построении такого правила используются определенные функции результатов наблюдений $g\left(x_1, x_2, x_3, \ldots x_N \right)$, называемые *статистиками для проверки гипотез.* Все возможные значения подобных статистик делятся на две части: *область принятия гипотезы* и *критическую область.* Проверка гипотезы сводится к выяснению того, попадает или нет конкретное значение статистики, вычисленное по выборке, в критическую область: если нет — гипотеза принимается как не противоречащая результатам наблюдения, если да — гипотеза отвергается. При этом всегда возможно совершить ошибку; различные типы возможных ошибок приведены в табл. 2.

Таблица 2. Виды ошибок при проверке статистических гипотез

Гипотеза	Объективно верна	Объективно неверна
Принимается	Правильное решение	Ошибка II рода
Отвергается	Ошибка 1 рода	Правильное решение

Вероятность совершить ошибку I рода называется *уровнем значимости критерия* и обозначается *q*. Обычно уровень значимости выбирают равным 0,01; 0,1; 0,05 (последнее значение - наиболее часто).

Критерии значимости - это критерии, с помощью которых проверяют гипотезы об абсолютных значениях параметров или о соотношениях между ними для генеральных совокупностей с известной (с точностью до параметров) функцией распределения вероятностей.

Для пояснения идеи построения критериев значимости предположим, что некоторая оценка $\hat{\Theta}$, используемая затем в качестве статистики *g*, вычислена по выборке объема *N*. Пусть имеются причины считать, что истинное значение оцениваемого параметра Θ , т. е. его значение в генеральной совокупности, равно Θ_0 . Это проверяемое предположение часто называют *нулевой гипотезой* H_0 и пишут $H_0 : \Theta = \Theta_0$.

Даже если нулевая гипотеза справедлива, то выборочное значение $\hat{\Theta}$ обычно не совпадает точно с Θ_0 , поскольку оно является всего лишь одним из конкретных значений случайной величины $\hat{\Theta}$, порожденной всевозможными выборками объема *N*. Спрашивается: насколько сильно $\hat{\Theta}$ должно отличаться от Θ_0 , чтобы в достаточной мере обоснованно можно было отвергнуть нулевую гипотезу? Если известна функция плотности вероятности оценки $f(\hat{\Theta})$, построенная теоретически в предположении справедливости нулевой гипотезы, то с ее помощью несложно найти такую зону, вероятность случайных попаданий в которую (когда $\boldsymbol{H_0}$

верна) мала (равна малому значению *q).* Эта зона и может использоваться в качестве критической области критерия.

Вид критической области полностью определяется характером *альтернативной гипотезы* $\boldsymbol{H_1}$, т. е. гипотезы, противопоставляемой нулевой, той гипотезы, в пользу которой склоняется исследователь, отвергая проверяемую нулевую гипотезу.

Если нулевой гипотезе $H_0:\Theta=\Theta_0$ противопоставляется альтернативная гипотеза $H_0:\Theta\neq\Theta_0$, то критерий для проверки $\boldsymbol{H_0}$ носит название *двустороннего,* а его критическая область состоит из двух частей. Если же альтернативная гипотеза формулируется в виде $H_1:\Theta<\Theta_0$ или $H_1:\Theta>\Theta_0$, то соответствующие критерии называются *односторонними* и их критические области содержат всего одну часть.

При проверке статистических гипотез используются два метода. Первый метод состоит в сравнении статистики с критическим значением квантиля, соответствующего данной статистике теоретического распределения. При этом нулевая гипотеза принимается, если значение статистики не превышает критического значения квантиля распределения. Второй метод заключается в вычислении p-value, представляющего собой пороговое значение вероятности распределения статистики, ниже которого нулевая гипотеза отвергается. Малое вычисленное значение p-value (обычно менее 0.05) считается маловероятным событием, что дает основание отвергнуть нулевую гипотезу. Удобство использования p-value в качестве инструмента проверки гипотез связано с тем, что при использовании первого метода по величине статистики трудно сразу сказать, превышает оно критическое значение или нет, поскольку для разных статистик критические значения квантилей распределения различны. В то же время, вычисленное значение p-value позволяет сразу сделать вывод о принятии нулевой гипотезы, поскольку критическое значение (обычно 0.05) в данном случае всегда одно для разных распределений.

На практике приходится иметь дело с проверкой следующего типа гипотез:

- гипотеза о предполагаемом виде закона распределения случайной величины.
- гипотеза равенства двух дисперсий случайной величины;
- гипотеза равенства средних;

Методы проверки статистических гипотез можно разделить на две группы: *параметрические* и *непараметрические*, называемые также методами, *свободными от распределения*. В методах первой группы предполагается, что данные следуют определенному вероятностному распределению (например, нормальному) с неизвестными параметрами (математическим ожиданием и дисперсией). Методы второй группы, напротив, не требуют предположений о виде распределения данных. Они основаны на рангах или частотах наблюдений, а не на конкретных значениях. Параметрические методы являются более мощными и точными, однако их применение ограничено случаями, когда выполняется предположение о виде распределения анализируемых случайных величин (обычно это распределение предполагается нормальным). Непараметрические методы свободны от этого требования, поэтому область их применения значительно шире.

3.1.3.1. Параметрические методы проверки гипотез

Сравнение двух дисперсий. При обработке результатов наблюдений часто возникает необходимость сравнить две или несколько выборочных дисперсий. Основная гипотеза, которая при этом проверяется, - можно ли считать две сравниваемые выборочные дисперсии оценками одной и той же генеральной дисперсии.

Пусть на основании двух выборок объемом N_1 и N_2 получены оценки дисперсий s_1^2 и s_2^2 соответственно. Предположим, что первая дисперсия сделана из генеральной совокупности с дисперсией σ_1^2, а вторая - из генеральной совокупности с дисперсией σ_2^2. Проверяется нулевая гипотеза о равенстве генеральных дисперсий $H_0: \sigma_1^2 = \sigma_2^2$. Чтобы отвергнуть эту гипотезу, нужно доказать значимость различия между s_1^2 и s_2^2 при выбранном уровне значимости *q*. В качестве критерия значимости обычно используется критерий Фишера (*F*-критерий). В соответствии с данным критерием различие между дисперсиями считается незначимым, если выполняется неравенство

$$g=\frac{s_1^2}{s_2^2} \quad , \quad g \le F(l_1, l_2, q) \quad , \tag{100}$$

где $l_1 = N_1 - 1$ - число степеней свободы первой выборки, $l_2 = N_2 - 1$ - число степеней свободы второй выборки; в числителе дроби стоит большая из сравниваемых дисперсий.

Значение критерия Фишера (квантиль распределения Фишера для доверительной вероятности *p*=1-*q*) можно вычислить с помощью функции scipy.stats.f.ppf(q, dfn, dfd). Значение p-value вычисляется как 1 - scipy.stats.f.cdf(g, l_1, l_2).

Сравнение значимости различия двух средних. Для сравнения между собой двух средних, полученных по выборкам из нормально распределенных генеральных совокупностей, применяется критерий Стьюдента, или *t*-критерий. Статистика для сравнения двух средних $\bar{x}$ и $\bar{y}$, полученных на основании выборок с объемами N_1 и N_2 , соответственно, имеет вид

$$g=\frac{|\bar{x}-\bar{y}|}{s\sqrt{\frac{1}{N_1}+\frac{1}{N_2}}} \quad , \tag{101}$$

где *s* - среднеквадратическое отклонение. При этом имеется в виду, что различие между дисперсиями s_1^2 и s_2^2 незначимо, так что в качестве среднеквадратического отклонения можно принять оценку

$$s=\sqrt{\frac{s_1^2+s_2^2}{2}} \quad . \tag{102}$$

В противном случае в качестве *s* можно принять корень из средневзвешенного значения выборочных дисперсий.

Различие между $\bar{x}$ и $\bar{y}$ считается незначимым, если выполняется неравенство

$$g \le t\left(\frac{q}{2}, l\right) \quad , \tag{103}$$

где $t\left(\frac{q}{2}, l\right)$ - значение критерия Стьюдента (квантиль распределения Стьюдента для доверительной вероятности $p = 1 - \frac{q}{2}$, l - число степеней свободы, принимаемое равным $l = N_1 + N_2 - 2$.

При одностороннем критерии уровень значимости вместо $\frac{q}{2}$ необходимо брать равным *q*. Значение критерия Стьюдента можно вычислить с помощью функции scipy.stats.t.ppf(q, df).

Сравнение выборочного распределения и распределения генеральной совокупности. Для проверки гипотезы о предполагаемом виде распределения применяют *критерии согласия*. Например, часто приходится проверять гипотезу о нормальном виде распределения. Среди различных критериев согласия наиболее употребителен критерий согласия χ^2 .

Проверку гипотезы о виде функции распределения с помощью этого критерия производят следующим образом:

1. По выборке строят гистограмму с числом интервалов группирования значений случайной величины K и вычисляют оценки параметров проверяемой функции распределения.
2. Определяют теоретическую вероятность p_m попадания случайной величиной в каждый из интервалов (значения параметров распределения при этом заменяют их оценками).
3. Рассчитывают статистику вида

$$g = N \sum_{m=1}^{K} \frac{\left(\frac{N_m}{N} - p_m \right)^2}{p_m} \quad , \tag{104}$$

где N_m - количество попаданий случайной величины в *m*-й интервал.

4. Если вычисленное значение статистики удовлетворяет неравенству

$$g \le \chi^2(l, q) \quad , \tag{105}$$

где $\chi^2(l, q)$ - квантиль распределения χ^2 с числом степеней свободы $l = K - r - 1$ и уровнем значимости *q*, то гипотеза о принадлежности проверяемого распределения выбранному типу принимается. Квантиль распределения χ^2 можно вычислить с помощью функции scipy.stats.chi2.ppf(q, df).

Пример 5. Проверка статистической гипотезы о принадлежности выборки нормальному закону распределения. Для проверки гипотезы нормальности с помощью критерия хи-квадрат в Python можно использовать библиотеку scipy.stats. Ниже приведен пример программы, которая генерирует выборки из нормального, равномерного и экспоненциального распределений, а затем применяет критерий хи-квадрат для проверки нормальности.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
# Функция для проверки нормальности с помощью критерия хи-квадрат
def chi_square_normality_test(data, num_bins=10, alpha=0.05):
    # Создаем гистограмму выборки
```

```
    observed_freq, bin_edges = np.histogram(data, bins=num_bins)
    # Находим центры интервалов разбиения выборки
    bin_centers = 0.5 * (bin_edges[1:] + bin_edges[:-1])
    # Оценка параметров нормального распределения
    mu, std = np.mean(data), np.std(data)
    # Ожидаемая частота для нормального распределения
        expected_freq = np.array([stats.norm(mu, std).cdf(bin_edges[i+1]) - stats.norm(mu, std).cdf(bin_edges[i]) for i in range(num_bins)]) * len(data)
    # Нормализация ожидаемых частот
    expected_freq = expected_freq / np.sum(expected_freq) * np.sum(observed_freq)
    # Применяем критерий хи-квадрат
    chi2_stat, p_value = stats.chisquare(observed_freq, expected_freq)
    # Определяем квантиль распределения хи-квадрат
    degrees_of_freedom = num_bins - 1  # Степени свободы
    critical_value = stats.chi2.ppf(1 - alpha, degrees_of_freedom)
    return chi2_stat, p_value, critical_value
# Генерация выборок
np.random.seed(0)  # Для воспроизводимости
sample_size = 1000
normal_sample = np.random.normal(loc=0, scale=1, size=sample_size)
uniform_sample = np.random.uniform(low=-np.sqrt(3), high=np.sqrt(3), size=sample_size)
exponential_sample = np.random.exponential(scale=1, size=sample_size)
# Проверка нормальности для каждой выборки
samples = {
    "Normal": normal_sample,
    "Uniform": uniform_sample,
    "Exponential": exponential_sample
}
for label, sample in samples.items():
    chi2_stat, p_value, critical_value = chi_square_normality_test(sample)
     print(f"{label} Sample: Chi-square Statistic = {chi2_stat:.4f}, p-value = {p_value:.4f}, Critical Value = {critical_value:.4f}")
```

Результаты расчета:

Normal Sample: Chi-square Statistic = 7.0116, p-value = 0.6359, Critical Value = 16.9190

Uniform Sample: Chi-square Statistic = 227.8705, p-value = 0.0000, Critical Value = 16.9190

Exponential Sample: Chi-square Statistic = 811133490647.2021, p-value = 0.0000, Critical Value = 16.9190

По результатам расчета можно сделать следующие выводы:

1. Нормальное распределение. Статистика хи-квадрат (7.0116) меньше критического значения (16.9190), а p-value = 0.6359, что существенно больше 0.05. Это подтверждает, что мы не можем отвергнуть нулевую гипотезу о нормальности данных. Результаты теста согласуются с тем, что выборка действительно нормальная.

2. Равномерное распределение. Статистика хи-квадрат (227.8705) значительно превышает критическое значение (16.9190), а p-value = 0.0000 (до четвертого знака, включительно), что меньше 0.05. Это подтверждает, что в данном случае мы не можем принять нулевую гипотезу о нормальности данных. Результаты согласуются с тем, что выборка не является нормальной.

3. Экспоненциальное распределение. Статистика хи-квадрат (811133490647.2021) также значительно превышает критическое значение (16.9190), а p-value = 0.0000 (до четвертого знака, включительно). В этом случае мы снова отвергаем нулевую гипотезу о нормальности данных, что соответствует тому, что выборка не является нормальной.

Таким образом, используя как p-value, так и сравнение статистики с критическим значением, мы можем уверенно делать выводы о нормальности выборок. В данном случае, нормальная выборка соответствует нормальному распределению, в то время как равномерная и экспоненциальная выборки явно не соответствуют ему.

Критерий проверки гипотез хи-квадрат является универсальным методом и пригоден для проверки принадлежности выборки к любому распределению. Вместе с тем, на практике чаще всего возникает задача проверки гипотезы нормальности распределения, то есть соответствия выборки распределению Гаусса. Для проверки гипотезы нормальности по сравнению с использованием критерия хи-квадрат более эффективен критерий Шапиро-Уилка [24]. Данный критерий основан на сравнении порядковых статистик выборки с ожидаемыми порядковыми статистиками нормального распределения. Статистика теста вычисляется как отношение оптимальной линейной несмещенной оценки дисперсии к её обычной оценке методом

максимального правдоподобия. Метод Шапиро–Уилка обладает высокой мощностью, особенно при небольших объемах выборки (до 50 наблюдений), что делает его предпочтительным выбором для проверки нормальности в таких случаях. Метод Шапиро-Уилка реализован в библиотеке SciPy для Python. Функция scipy.stats.shapiro() позволяет выполнить тест Шапиро-Уилка на нормальность распределения данных. Следующая простая программа иллюстрирует применение теста Шапиро-Уилка для проверки гипотезы нормальности выборки.

```
from scipy.stats import shapiro
import numpy as np
def shapiro_test(data):
    stat, p = shapiro(data)
    print('Статистика теста: %.3f' % stat)
    print('p-значение: %.3f' % p)
    if p > 0.05:
        print('Выборка, вероятно, имеет нормальное распределение')
    else:
        print('Выборка, вероятно, не имеет нормального распределения')
# Сгенерируем нормально распределенные и равномерно распределенные данные
data_n = np.random.normal(0, 1, 100)
data_u = np.random.rand(100)
data={"Нормальная выборка":data_n,"Равномерная выборка":data_u}
# Выполним тест Шапиро-Уилка
for key in data.keys():
    print(key)
    shapiro_test(data[key])
```

Результат вычислений:

```
Нормальная выборка
Статистика теста: 0.986
p-значение: 0.351
Выборка, вероятно, имеет нормальное распределение
```

Равномерная выборка
Статистика теста: 0.943
p-значение: 0.000
Выборка, вероятно, не имеет нормального распределения

Как видно, в данном примере программа правильно различает выборки нормально и равномерно распределенных случайных чисел.

При выполнении гипотезы о нормальном распределении двух или нескольких выборок случайных величин можно проверить гипотезы о равенстве их параметров. Обычно проверяются гипотезы о равенстве дисперсий двух распределений и их математических ожиданий. Ниже приведен пример простой программы, в которой генерируются две выборки нормально распределенных случайных величин с разными значениями параметров и производится сравнение их дисперсий и математических ожиданий с использованием критериев Фишера и Стьюдента.

```
import numpy as np
from scipy.stats import f, t
# Генерация данных
np.random.seed(42)
sample1 = np.random.normal(10, 2, 50)
sample2 = np.random.normal(12, 3, 40)
# Проверка гипотезы о равенстве дисперсий
dispers=[np.var(sample1,ddof=1),np.var(sample2,ddof=1)]
F_stat = max(dispers) / min(dispers)
p_value = 1 - f.cdf(F_stat, len(sample1)-1, len(sample2)-1)
print(f"p-value для F-статистики: {p_value}")
alpha = 0.05
if p_value < alpha:
    print("Отвергаем гипотезу о равенстве дисперсий.")
else:
    print("Не отвергаем гипотезу о равенстве дисперсий.")
# Проверка гипотезы о равенстве математических ожиданий
```

```
pooled_var = ((len(sample1)-1)*np.var(sample1) + (len(sample2)-1)*np.var(sample2)) / (len(sample1)+len(sample2)-2)
SE = np.sqrt(pooled_var * (1/len(sample1) + 1/len(sample2)))
t_stat = (np.mean(sample1) - np.mean(sample2)) / SE
p_value = 2 * (1 - t.cdf(abs(t_stat), len(sample1)+len(sample2)-2))
print(f"p-value для t-статистики: {p_value}")
if p_value < alpha:
    print("Отвергаем гипотезу о равенстве математических ожиданий.")
else:
    print("Не отвергаем гипотезу о равенстве математических ожиданий.")
```

В данной программе генерируются две выборки sample1 и sample1 нормально распределенных случайных величин с параметрами $(m_x=10, \sigma_x^2=2, N=50)$ и $(m_x=12, \sigma_x^2=3, N=40)$, соответственно. Далее вычисляется статистика для распределения Фишера (100) и соответствующее значение p_value. Проверка гипотезы равенства дисперсий осуществляется сравнением вычисленного p_value с заданным уровнем значимости alpha. На основе вычисленных дисперсий рассчитывается их средневзвешенное значение pooled_var и, с его использованием, статистика (101). Для средних значений вычисляется соответствующее p_value с использованием t-распределения, которое затем также сравнивается с критическим значением. На печать выводятся результаты проверки гипотез равенства дисперсий и равенства математических ожиданий.

Для значений параметров, заданных в тексте, программа выдает правильный результат о значимости различия дисперсий и математических ожиданий:

```
p-value для F-статистики: 0.006087425098451393
Отвергаем гипотезу о равенстве дисперсий.
p-value для t-статистики: 5.041909854064386e-07
Отвергаем гипотезу о равенстве математических ожиданий.
```

Заметим, что правильность выводов критическим образом зависит от объема выборок. Например, если в данной программе сохранить значения параметров прежними, а объем выборок уменьшить в 10 раз, то оба вывода окажутся неверными:

```
p-value для F-статистики: 0.14149273150295139
```

Не отвергаем гипотезу о равенстве дисперсий.

p-value для t-статистики: 0.10789269247419253

Не отвергаем гипотезу о равенстве математических ожиданий.

То есть в этом случае совершается ошибка второго рода: объективно неверная гипотеза принимается (см. табл. 2).

Анализ выбросов многомерных распределений. Если одномерная случайная величина Y распределена по закону $N(\mu,\sigma^2)$, то случайная величина $(Y-\mu)^2/\sigma^2$ имеет распределение $\chi^2(1)$. В многомерном случае можно показать, что если случайный вектор X с p компонентами имеет многомерное нормальное распределение с вектором средних μ и матрицей ковариации Σ, то величина

$$\chi^2=(x-\mu)^T\Sigma^{-1}(x-\mu) \qquad (106)$$

имеет распределение $\chi^2(p)$. Если μ и Σ известны, то эта статистика может быть использована для проверки возможной аномальности наблюдаемого вектора x, т.е. наличия выбросов у его компонент. Здесь в качестве критерия проверки нулевой гипотезы о незначимости различия вектора x от его математического ожидания μ является квантиль распределения χ^2. Если вычисленное значение критерия оказалось больше критического значения χ^2_{cr} для выбранного заранее уровня значимости q, то наблюдаемый вектор можно считать аномальным, и его координаты должны быть проверены на наличие ошибок.

В большинстве случаев параметры μ и Σ неизвестны, и поэтому использование статистики χ^2 вида (106) не обосновано. Имеется другая процедура проверки, которая использует статистику, являющуюся выборочным аналогом выражения (106). Пусть $x_1,\dots x_k$ -

случайная выборка, имеющая распределение $N(\mu, \Sigma)$. Тогда выборочное среднее и ковариационная матрица имеют соответственно вид

$$\bar{x} = \frac{1}{k} \sum_{i=1}^{k} x_i \quad , \tag{107}$$

$$S = \frac{1}{k-1} \sum_{i=1}^{k} (x_i - \bar{x})(x_i - \bar{x})^T \quad . \tag{108}$$

Если x - некоторый вектор наблюдений, имеющий распределение $N(\mu, \Sigma)$, то выборочный аналог величины (106), называемый *выборочным расстоянием Махалонобиса*, дается формулой

$$D^2 = (x - \bar{x})^T S^{-1} (x - \bar{x}) \quad . \tag{109}$$

Можно показать, что величина

$$F = \frac{(k-p)k}{(k^2-1)p} D^2 \tag{110}$$

имеет *F*-распределение с *p* и $k-p$ степенями свободы.

Процедура проверки на наличие выбросов среди наблюдений использует статистику, задаваемую выражением (110), где $\bar{x}$ и S вычисляются по подмножеству векторов той же выборки, уже проверенных на выбросы.

Приведем процедуру проверки, примененную к случайной выборке $x_1,\dots x_n$ объема *n*.

1. Для каждого вектора наблюдений x_i , $i=1,\dots,n$ вычисляется выборочный вектор средних $\bar{x}_i$ и ковариационная матрица S_i по всем $k=n-1$ векторам наблюдений, исключая x_i . Согласно выражению (109) вычисляется выборочное расстояние Махалонобиса D_i^2 между x_i и $\bar{x}_i$ с использованием ковариационной матрицы S_i . Затем с помощью формулы (110) вычисляются F_i для $k=n-1$ и соответствующее критическое значение F_{icr} для выбранного уровня значимости q .

2. Проверка $F_1,F_2,\dots F_n$. Если некоторые из $F_i>F_{icr}$, то вектор наблюдений, соответствующий наибольшему значению F , считается выбросом и исключается из выборки. Процедура повторяется для оставшихся $n-1$ наблюдений.

Проверка гипотез о векторах средних для многомерных распределений. Пусть Y_i , $i=1,2$ - случайные величины, распределенные по закону $N(\mu_i,\sigma^2)$, а $y_{i1},\dots,y_{in_i}$ - случайные выборки из этих распределений. Для проверки гипотезы $H_0:\mu_1=\mu_2$ против гипотезы $H_1:\mu_1\neq\mu_2$ при неизвестной дисперсии σ^2 можно использовать статистику

$$t=\frac{\bar{y}_1-\bar{y}_2}{\sqrt{s_p^2\left(n_1^{-1}+n_2^{-1}\right)}} \quad , \qquad (111)$$

где $\bar{y}_i$ есть i -е выборочное среднее, $i=1,2$ и s_p^2 - общая дисперсия. Гипотеза H_0 отвергается, если $|t|>t_{q/2}(n_1+n_2-2)$ для некоторого заранее выбранного уровня

значимости q . Многомерным аналогом этой двухвыборочной *t*-статистики Стьюдента является двухвыборочная *T^2-статистика Хотеллинга*. Предположим, что случайные векторы X_i имеют распределение $N(\mu_i, \Sigma)$, $i=1,2$. Пусть $x_{i1},\ldots,x_{in_i}$ - случайная выборка из *i*-го распределения. Матрица Σ оценивается объединенной выборочной ковариационной матрицей

$$S=\frac{1}{n_1+n_2-2}\left[(n_1-1)S_1+(n_2-1)S_2\right] \quad , \tag{112}$$

где $\mathbf{S}_i$ – стандартная оценка ковариационной матрицы по *i*-й выборке. Тогда двухвыборочная T^2-статистика имеет вид

$$T^2=\frac{n_1 n_2}{n_1+n_2}(\bar{x}_1-\bar{x}_2)^T S^{-1}(\bar{x}_1-\bar{x}_2) \quad , \tag{113}$$

где $\bar{x}_i=\frac{1}{n_i}\sum_{j=1}^{n_i}\bar{x}_{ij}$, $i=1,2$ - оценка μ_i .

Если гипотеза $H_0:\mu_1=\mu_2$ верна, то величина

$$F=\frac{n_1+n_2-p-1}{(n_1+n_2-2)p}T^2 \tag{114}$$

имеет *F*-распределение с *p* и n_1+n_2-p-1 степенями свободы. Вычисленное значение статистики *F* сравнивается с критическим значением F_{cr} для выбранного уровня значимости *q*. При выполнении равенства $F>F_{cr}$ гипотеза H_0 отвергается.

Кроме проверки гипотез о средних, могут быть построены многомерные аналоги доверительных интервалов для линейных комбинаций компонент вектора μ . Для заданного

набора констант $a_1,\ldots,a_p$ многомерный аналог доверительного интервала для $\sum_{i=1}^{p} a_i \mu_i$ имеет вид

$$\sum_{i=1}^{p} a_i \bar{x}_i \pm \left[\frac{(n-1)p}{(n-p)n} F_q(p, n-p) \sum_{i=1}^{p} \sum_{j=1}^{p} a_i s_{ij} a_j \right]^{\frac{1}{2}} \qquad (115)$$

В частном случае, многомерный аналог доверительного интервала для *i*-й компоненты вектора μ имеет вид

$$\bar{x}_i \pm \left[\frac{(n-1)p}{(n-p)n} F_q(p, n-p) s_i^2 \right]^{\frac{1}{2}} \quad , \quad i=1,\ldots,p \quad . \qquad (116)$$

Таким образом, можно получить доверительный интервал, использующий многомерную структуру данных, аналогичный тому, который мы получали с помощью распределения Стьюдента.

3.1.3.2. Непараметрические методы

Непараметрические методы в статистике представляют собой класс методов, которые не требуют выполнения строгих предположений о вероятностных распределениях анализируемых данных. Эти методы особенно полезны в ситуациях, когда известные распределения не могут быть применены или когда размер выборки мал. Традиционные методы проверки гипотез параметрической статистики основываются на предположении о нормальном характере распределения случайных величин в анализируемых выборках. Однако часто, особенно для малых выборок, тесты проверки выборки на нормальность показывают отрицательный результат. В других случаях, наоборот, тест проверки на нормальность может оказаться положительным, хотя исследуемая случайная величина по своему смыслу не может иметь нормального распределения, то есть принимается гипотеза, которая объективна не верна. В этом случае имеет место ошибка второго рода (см. табл. 2).

Одним из фундаментальных непараметрических методов является метод Колмогорова-Смирнова [25,26]. Этот метод используется для проверки гипотезы о том, что выборка происходит из определенного распределения, или для сравнения двух эмпирических распределений.

Пусть анализируемая случайная величина X имеет интегральную функцию распределения $F(x)$. На основании выборки объемом N анализируется эмпирическая (выборочная) функция распределения вида

$$F_N(x)=\frac{1}{N}\sum_{i=1}^{N} I_{X_i\leqslant x} \quad , \tag{117}$$

где $I_{X_i\leqslant x}=\begin{cases}1, X_i\leqslant x\\ 0, X_i>x\end{cases}$.

Выполняется проверка того, является ли эмпирическая функция $F_N(x)$ порожденной случайной величиной с функцией $F(x)$. Статистика критерия для эмпирической функции распределения $F_N(x)$ определяется следующим образом

$$D_N=\sup_{x\in R}|F_N(x)-F(x)| \quad , \tag{118}$$

где под sup понимается точная верхняя грань функции $|F_N(x)-F(x)|$.

Тогда по теореме Колмогорова для введённой статистики справедливо

$$\forall t>0:\lim_{N\to\infty} P(\sqrt{N}\,D_N\leq t)=K(t) \quad , \tag{119}$$

где

$$K(t)=\sum_{(j=-\infty)}^{(+\infty)}(-1)^j e^{(-2j^2t^2)} \tag{120}$$

- критерий, имеющий правостороннюю область.

Таким образом, $K(t)$ является предельным законом распределения случайной величины $\sqrt{N}\,D_N$ и позволяет проводить сравнение выборочного распределения с некоторым известным теоретическим распределением $F(x)$. В отличие от критерия хи-квадрат, основанного на вычислении частот, то есть аналогов дифференциальной функции распределения $f(x)$, критерий Колмогорова для построения статистики использует накопленные частоты, то есть аналог интегральной функции распределения $F(x)$. Мощность теста хи-квадрат зависит от размера выборки и числа категорий. Он показывает высокую мощность при большом количестве наблюдений и равномерном распределении данных по категориям. Однако его эффективность значительно снижается на малых выборках, где он может не обнаруживать различия между распределениями. Мощность теста Колмогорова также зависит от размера выборки, но он менее

чувствителен к малым выборкам по сравнению с традиционными параметрическими тестами, что делает его более подходящим для анализа небольших объемов данных.

В библиотеке scipy.stats для проверки гипотезы нормальности некоторой выборки data на основе критерия Колмогорова используется функция

statistic, p_value = kstest(data, 'norm').

Ниже приведен пример программы для сравнения эффективности теста Шапиро-Уилка и теста Колмогорова для выборок заданного объема из трех разных распределений: нормального, равномерного и хи-квадрат, которое при $df > 2$ выглядит похожим на нормальное (см. рис. 6).

```
import numpy as np
from scipy.stats import chi2, shapiro, kstest
def check_sp(stat,p):
    print('Статистика теста: %.3f' % stat)
    print('p-значение: %.3f' % p)
    if p > 0.05:
        print('Выборка, вероятно, имеет нормальное распределение')
    else:
        print('Выборка, вероятно, не имеет нормального распределения')
N=200
df=N-2
print(f"Объем выборки: {N}")
data={"Нормальное":np.random.normal(0, 1, N),
     "Равномерное":np.random.rand(N),
     "Хи-квадрат":chi2.rvs(df, size=N)}
for key in data.keys():
    print(f"Проверяем распределение: {key}")
    print("Тест Шапиро-Уилка")
    stat, p = shapiro(data[key])
    check_sp(stat,p)
    print("Тест Колмогорова")
    stat, p = kstest(data[key], 'norm')
```

```
check_sp(stat,p)
```

Ниже приведен вывод результатов для N=100:

```
Объем выборки: 100
Проверяем распределение: Нормальное
Тест Шапиро-Уилка
Статистика теста: 0.987
p-значение: 0.439
Выборка, вероятно, имеет нормальное распределение
Тест Колмогорова
Статистика теста: 0.100
p-значение: 0.254
Выборка, вероятно, имеет нормальное распределение
Проверяем распределение: Равномерное
Тест Шапиро-Уилка
Статистика теста: 0.957
p-значение: 0.002
Выборка, вероятно, не имеет нормального распределения
Тест Колмогорова
Статистика теста: 0.501
p-значение: 0.000
Выборка, вероятно, не имеет нормального распределения
Проверяем распределение: Хи-квадрат
Тест Шапиро-Уилка
Статистика теста: 0.953
p-значение: 0.001
Выборка, вероятно, не имеет нормального распределения
Тест Колмогорова
Статистика теста: 1.000
p-значение: 0.000
Выборка, вероятно, не имеет нормального распределения
```

При объеме выборок $N=100$ оба теста правильно различают все три распределения.

Однако при $N=20$ результат оказывается иным:

Объем выборки: 20
Проверяем распределение: Нормальное
Тест Шапиро-Уилка
Статистика теста: 0.976
p-значение: 0.877
Выборка, вероятно, имеет нормальное распределение
Тест Колмогорова
Статистика теста: 0.123
p-значение: 0.886
Выборка, вероятно, имеет нормальное распределение
Проверяем распределение: Равномерное
Тест Шапиро-Уилка
Статистика теста: 0.891
p-значение: 0.028
Выборка, вероятно, не имеет нормального распределения
Тест Колмогорова
Статистика теста: 0.514
p-значение: 0.000
Выборка, вероятно, не имеет нормального распределения
Проверяем распределение: Хи-квадрат
Тест Шапиро-Уилка
Статистика теста: 0.911
p-значение: 0.067
Выборка, вероятно, имеет нормальное распределение
Тест Колмогорова
Статистика теста: 1.000
p-значение: 0.000
Выборка, вероятно, не имеет нормального распределения

При объеме выборок $N=20$ тест Шапиро-Уилка правильно различает равномерное и нормальное распределения, но делает ошибку в случае распределения хи-квадрат, принимая его за

нормальное. Тест Колмогорова правильно различает все три вида распределений, что подтверждает его преимущество при работе с малыми выборками.

Теорема Смирнова касается сравнения двух выборочных распределений. Пусть $F_{1,N}(x)$, $F_{2,M}(x)$ — эмпирические функции распределения, построенные по независимым выборкам объёмами N и M, соответственно. Статистика для сравнения данных выборок определяется как

$$D_{N,M} = \sup_{x \in R} |F_N(x) - F_M(x)| \quad . \tag{121}$$

Согласно теореме Смирнова

$$\forall t > 0 : \lim_{N,M \to \infty} P\left(\sqrt{\frac{NM}{N+M}} D_{N,M} \le t\right) = K(t) \quad . \tag{122}$$

То есть $K(t)$ (120) является предельным законом распределения случайной величины $\sqrt{\frac{NM}{N+M}} D_{N,M}$ и может использоваться для проверки гипотезы H_0 об однородности двух выборок.

В библиотеке scipy.stats для проверки гипотезы однородности двух выборок data1 и data2 на основе критерия Колмогорова - Смирнова используется функция

statistic, p_value = ks_2samp(data1, data2).

Как параметрические критерии, так и критерий Колмогорова-Смирнова основаны на анализе значений случайной величины. Вместе с тем существует группа критериев, в которых собственно значения случайной величины не используются, а достаточно знать лишь их ранги в выборке. Ранг представляет собой индекс или номер случайной величины в упорядоченном по возрастанию списке. То есть для получения рангов нужно просто упорядочить (проранжировать) выборку по интересующему нас признаку: сделать из выборки вариационный ряд. Номер наблюдения в вариационном ряду и будет его рангом. Конечно, при переходе от значений случайной величин к их рангам мы теряем информацию о самих значениях. Но в некоторых случаях этого вполне достаточно, чтобы сравнить две выборки, а иногда переход к рангам является даже удобным.

К числу классических ранговых критериев является критерий Уилкоксона, предложенный им в 1945 г. [27]. Критерий предназначен для сопоставления показателей, измеренных в двух

разных условиях на одной и той же выборке испытуемых, то есть для анализа согласованных выборок. Например, это может быть сравнение результатов тестов до и после какого-либо воздействия на одну и ту же группу людей. Он применим в тех случаях, когда признаки измерены, по крайней мере, в порядковой шкале и рекомендуется для малых выборок (до 25 наблюдений), поскольку с увеличением объема выборки распределение критерия Уилкоксона быстро приближается к нормальному.

Статистика критерия Уилкоксона строится следующим образом. Для двух выборок x и y одинакового объема N строится ряд разностей $z_i=|x_i-y_i|$, который затем ранжируется по возрастанию. Далее вычисляются суммы рангов для положительных и отрицательных разностей: W^+ и W^-. Статистика критерия Уилкоксона W равна меньшей из сумм рангов: $W=min(W^+,W^-)$. Для проведения теста по критерию Уилкоксона в библиотеке SciPy имеется функция scipy.stats.wilcoxon. Ниже приведен пример программы, моделирующей проведение такого теста.

```
import numpy as np
from scipy.stats import wilcoxon
import matplotlib.pyplot as plt
# Генерация связанных выборок
np.random.seed(42)
n = 20  # Объем выборки
mu = 50  # Среднее значение
sigma = 10  # Стандартное отклонение
# Генерация данных до воздействия
before = np.random.normal(mu, sigma, n)
# Генерация данных после воздействия
after = before + np.random.normal(5, 2, n)  # Увеличение на 5 с небольшим шумом
# Визуализация связанных выборок
plt.figure(figsize=(10, 5))
plt.plot(before, label='До воздействия', marker='o')
plt.plot(after, label='После воздействия', marker='o')
plt.title('Связанные выборки: До и После воздействия')
```

```
plt.xlabel('Наблюдения')
plt.ylabel('Значения')
plt.legend()
plt.grid()
plt.show()
# Применение теста Вилкоксона
stat, p_value = wilcoxon(before, after)
print("\nРезультаты теста Вилкоксона:")
print("Статистика теста:", stat)
print("p-значение:", p_value)
if p_value < 0.05:
    print("Отвергаем нулевую гипотезу: выборки различаются")
else:
    print("Не отвергаем нулевую гипотезу: выборки не различаются")
```

Результаты работы программы:

```
Результаты теста Вилкоксона:
Статистика теста: 0.0
p-значение: 1.9073486328125e-06
Отвергаем нулевую гипотезу: выборки различаются
```

Графики выборок приведены на рисунке 12.

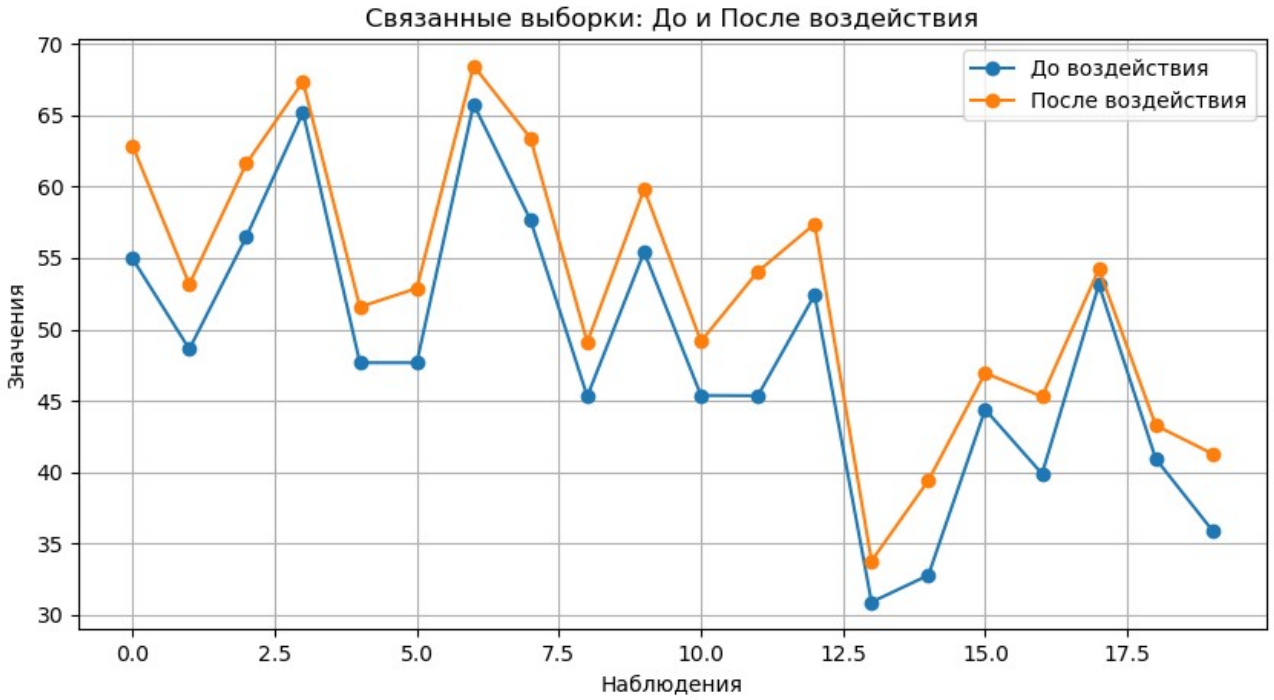

Рисунок 12. Графики связанных выборок для проверки по критерию Уилкоксона

Развитием критерия Уилкоксона для несвязанных выборок явился критерий Манна-Уитни [28], предложенный вскоре после его работы. Данный критерий, в отличие от критерия Уилкоксона, предназначен для сравнения несвязанных выборок.

Пусть $x_1,\dots,x_N$ и $y_1,\dots,y_M$ - две упорядоченные по возрастанию выборки. Для проверки гипотезы сдвига Манн и Уитни предложили ранговый критерий, основанный на статистике

$$U=\sum_{i=1}^{N}\sum_{j=1}^{M}h_{ij}\quad,\quad h_{ij}=\begin{cases}1, x_i<y_j\\ 0, x_i>y_j\end{cases}. \tag{123}$$

Здесь U - точное число пар значений x_i и y_j, для которых $x_i<y_j$.

Для проведения теста в библиотеке scipy.stats имеется функция mannwhitneyu. Ниже приведен текст программы, в котором генерируются две несвязанные выборки нормально распределенных случайных величин и проводится их сравнение методом Манна-Уитни.

```
import numpy as np
from scipy.stats import mannwhitneyu
```

```
import matplotlib.pyplot as plt
# Генерация несвязанных выборок
np.random.seed(24)
m1 = 15  # Объем первой выборки
m2 = 20  # Объем второй выборки
mu_group1 = 10  # Среднее значение первой группы
mu_group2 = 15  # Среднее значение второй группы
sigma_group1 = 10  # Стандартное отклонение первой группы
sigma_group2 = 12  # Стандартное отклонение второй группы
# Генерация данных
group1 = np.random.normal(mu_group1, sigma_group1, m1)
group2 = np.random.normal(mu_group2, sigma_group2, m2)
# Визуализация связанных выборок
plt.figure(figsize=(10, 5))
plt.plot(group1, label='Группа 1', marker='o')
plt.plot(group2, label='Группа 2', marker='o')
plt.title('Несвязанные выборки')
plt.xlabel('Наблюдения')
plt.ylabel('Значения')
plt.legend()
plt.grid()
plt.show()
# Применение теста Манна-Уитни
stat, p_value = mannwhitneyu(group1, group2)
print("\nРезультаты теста Манна-Уитни:")
print("Статистика теста:", stat)
print("p-значение:", p_value)
if p_value < 0.05:
    print("Отвергаем нулевую гипотезу: выборки различаются")
else:
    print("Не отвергаем нулевую гипотезу: выборки не различаются")
```

Ниже приведен вывод результатов теста и графики, иллюстрирующие смоделированные выборки (рис. 13).

Результаты теста Манна-Уитни:

Статистика теста: 63.0

p-значение: 0.003934908115431982

Отвергаем нулевую гипотезу: выборки различаются

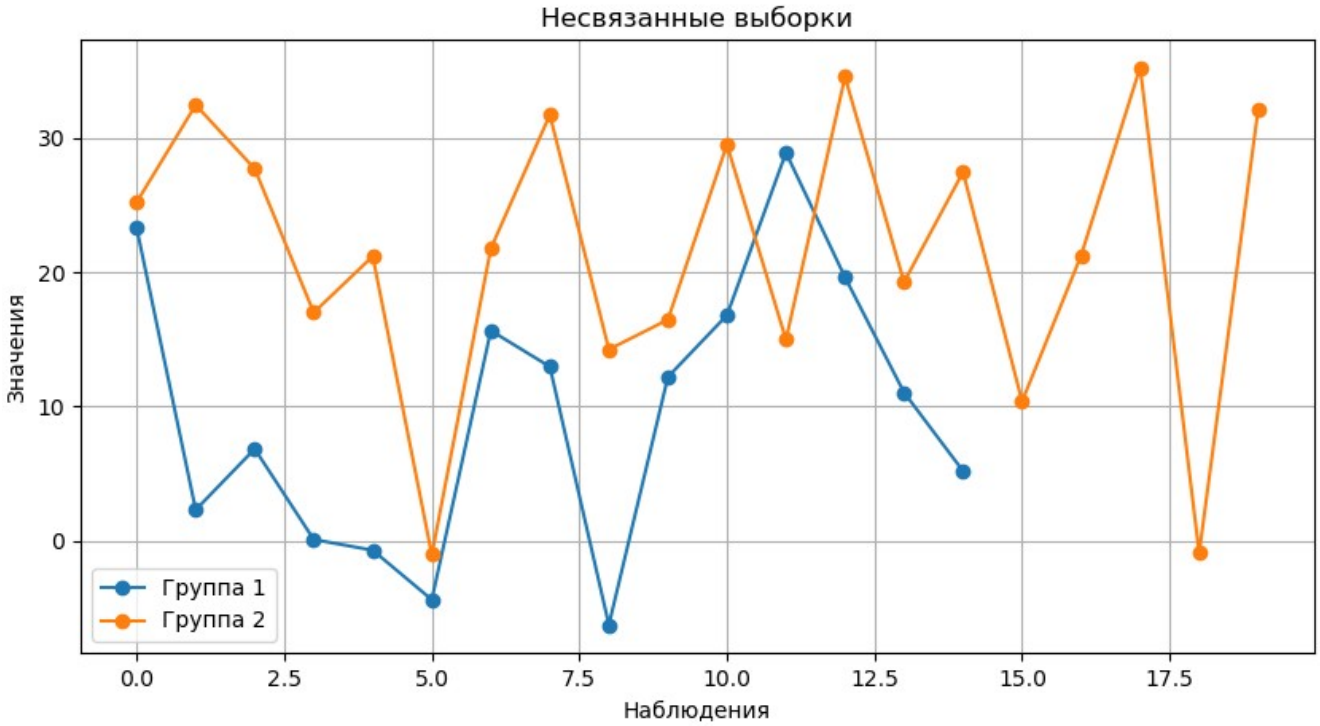

Рисунок 13. Графики выборок данных для проведения теста Манна-Уитни

Как видно, несмотря на малый объем выборок, тест Манна-Уитни правильно устанавливает их различие.

Часто возникает необходимость сравнения не двух, а сразу нескольких независимых выборок. Для этой цели можно использовать метод Краскала-Уоллиса, позволяющий определить, есть ли статистически значимые различия в медианах между несколькими группами. Данный тест представляет собой многомерное обобщение двухвыборочного критерия Манна-Уитни. Нулевая гипотеза в данном случае (H_0) формулируется следующим образом: медианы всех групп равны (или выборки происходят из одинаковых распределений). Альтернативная гипотеза (H_1): по

крайней мере одна из медиан групп отличается от других (или выборки происходят из разных распределений). Расчет статистики теста H осуществляется по формуле

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{k} \frac{R_i^2}{n_i} - 3(N+1) \quad , \tag{124}$$

где:

- N — общее количество наблюдений,
- R_i — сумма рангов для группы i ,
- n_i — количество наблюдений в группе i ,
- k — количество групп.

Для проведения анализа в библиотеке scipy.stats имеется функция kruskal. Ниже приведен текст программы, иллюстрирующий ее использование.

```
import numpy as np
from scipy.stats import kruskal
np.random.seed(24)
# Создаем данные для трех групп
mu=[5,7,8]
sig=[2,3,4]
N=[15,18,20]
group=[]
for i in range(3):
    group.append(np.random.normal(mu[i], sig[i], N[i]))
# Проводим тест Краскала-Уоллиса
statistic, p_value = kruskal(*group)
# Выводим результаты
print(f"Статистика теста: {statistic:.2f}")
print(f"p-значение: {p_value:.4f}")
# Интерпретируем результаты
alpha = 0.05
if p_value < alpha:
```

```
    print("Существуют статистически значимые различия между группами.")
else:
    print("Нет статистически значимых различий между группами.")
```

В данной программе создаются списки mu, sig, N, содержащие, соответственно, математические ожидания, стандартные отклонения и объемы трех выборок случайных величин с нормальным распределением, для которых далее выполняется проверка однородности с использованием статистики Краскала-Уоллиса H (124), результаты которой выводятся на печать. Ниже приведен пример вывода программы.

```
Статистика теста: 13.34
p-значение: 0.0013
Существуют статистически значимые различия между группами.
```

Ограничением данного теста является то, что он предполагает, что распределения в группах имеют схожую форму (симметричность не является строгим требованием, но может повлиять на результаты).

3.2. Дисперсионный анализ

В любом эксперименте средние значения наблюдаемых величин меняются в связи с изменением основных факторов, определяющих условия опыта (входных переменных), а также под воздействием случайных факторов. Входные переменные могут быть управляемыми (например, состав композиции) и неуправляемыми (например, физико-химические характеристики компонентов), поддающимися количественному описанию (например, содержание ингредиента в смеси) и не поддающимися количественному описанию - качественные факторы (например, торговая марка ингредиента). В любом случае цель эксперимента, как правило, состоит в том, чтобы оценить влияние входных переменных по отношению к случайным факторам. Для решения этой задачи может быть использован метод *дисперсионного анализа* [29] (ДА, в английской литературе используется аббревиатура ANOVA - сокращение от Analysis of variance). В основе данного метода лежит идея отделения дисперсии, приписываемой одной группе причин от дисперсии, приписываемой другим группам. В дисперсионном анализе используется свойство аддитивности дисперсии случайной величины, обусловленной действием независимых факторов.

Дисперсионный анализ пригоден для исследования любых факторов - качественных и количественных, управляемых и неуправляемых. Его преимущества особенно проявляются при

одновременном исследовании нескольких факторов (многофакторный дисперсионный анализ), поскольку в этом случае можно оценить не только влияние каждого фактора в отдельности, но и их взаимодействие, т. е. определить, как изменяется влияние одного фактора при изменении других. Метод дисперсионного анализа был предложен в 20-х годах XX столетия американским статистиком Р.А. Фишером и затем был развит Йейтсом. Существует несколько видов ДА, включая однофакторный (one-way ANOVA), многофакторный (factorial ANOVA) и повторные измерения (repeated measures ANOVA). Однофакторный ДА используется для анализа одного независимого переменного, тогда как многофакторный ДА позволяет исследовать взаимодействия между несколькими независимыми переменными.

3.2.1. Идея дисперсионного анализа

Пусть имеется некоторый фактор А, влияние которого необходимо оценить на отклик *y*. Фактор А изменяется (варьируется) на *k* уровнях. Для того чтобы оценить изменение отклика *y* под влиянием фактора А, удобно ввести величину, называемую дисперсией фактора А:

$$\sigma_A^2 = \frac{1}{k}\sum_{j=1}^{k}\left(y_j - \bar{y}\right)^2 \quad , \tag{125}$$

где $\bar{y}$ - среднее значение отклика по k уровням фактора А.

Несмотря на то, что фактор А не является случайной величиной, оценивать его влияние по величине σ_A^2 удобно в связи с тем, что данная величина вычисляется аналогично дисперсии случайной величины, и, таким образом, также характеризует рассеивание отклика, но обусловленное не случайными причинами, а влиянием входной переменной.

Рассмотрим идею дисперсионного анализа на примере изучения влияния одного фактора А на фоне случайных погрешностей, когда дисперсия воспроизводимости (ошибки) σ_ε^2 известна. При варьировании фактора А на k уровнях в результате наблюдения получим значения отклика $y_1, y_2, .. y_j, .. y_k$, рассеяние которых можно характеризовать выборочной дисперсией

$$s_o^2 = \frac{1}{k-1}\sum_{j=1}^{k}\left(y_j - \bar{y}\right)^2 \quad . \qquad (126)$$

Если отличие s_o^2 от σ_ε^2 незначимо, то разброс наблюдений, который она характеризует, связан со случайными факторами, и влияние фактора А несущественно. Если же отличие s_o^2 от σ_ε^2 значимо, то повышенный разброс наблюдений вызывается не только случайными причинами, но и влиянием фактора А, которое теперь нужно признать существенным. Так как в последнем случае складывается влияние двух независимых факторов: 1) случайных причин с дисперсией отклика σ_ε^2 ; 2) фактора А с дисперсией σ_A^2 , что приводит к общему рассеиванию наблюдений, - то общая дисперсия является суммой двух указанных, а ее оценка имеет вид

$$s_o^2 \approx \sigma_\varepsilon^2 + \sigma_A^2 \quad , \qquad (127)$$

откуда дисперсия фактора А определяется выражением

$$\sigma_A^2 \approx s_o^2 - \sigma_\varepsilon^2 \quad . \qquad (128)$$

В общем случае, когда дисперсия воспроизводимости не известна, схема ДА позволяет найти ее оценку наряду с оценками изучаемых факторов. С этой целью планируется проведение серий параллельных опытов при каждом из всех возможных сочетаний уровней изучаемых факторов.

Таким образом, идея ДА состоит в разложении оценки общего рассеивания отклика на составляющие, зависящие: 1) от случайных причин; 2) от каждого из рассматриваемых факторов; 3) от их взаимодействий в отдельности, а также в оценивании статистической значимости дисперсий последних относительно дисперсии воспроизводимости опыта. Для корректного применения ДА необходимо, чтобы данные соответствовали определенным предположениям:

нормальное распределение в каждой группе, равенство дисперсий (гомоскедастичность) и независимость наблюдений.

3.2.2. Однофакторный дисперсионный анализ

Пусть изучается влияние только одного фактора А, который варьируется на k уровнях. На каждом из уровней производится по n параллельных наблюдений. Таким образом, общий объем выборки $N = k \cdot n$. Результаты опытов можно представить в виде таблицы (см. табл. 3).

Таблица 3. Таблица опытов при однофакторном ДА

a_1	a_2	…	a_l	…	a_k
y_{11}	y_{12}	…	y_{1l}	…	y_{1k}
y_{21}	y_{22}	…	y_{21}	…	y_{1k}
…	…	…	…	…	…
y_{j1}	y_{j2}	…	y_{jl}	…	y_{jk}
…	…	…	…	…	…
y_{n1}	y_{n2}	…	y_{nl}	…	y_{nk}
$\bar{y}_1$	$\bar{y}_2$	…	$\bar{y}_l$	…	$\bar{y}_k$

При таком расположении откликов в таблице рассеивание между столбцами обуславливается влиянием фактора А, рассеивание внутри столбцов - влиянием случайных факторов. Вычислим средние арифметические $\bar{y}_l$ для каждого *l*-го уровня фактора:

$$\bar{y}_l = \frac{1}{n}\sum_{j=1}^{n} y_{jl} \quad . \tag{129}$$

Действие фактора случайности проявляется в рассеивании откликов на каждом уровне фактора А относительно среднего $\bar{y}_l$. Оценить фактор случайности можно, вычислив дисперсии воспроизводимости на каждом уровне фактора А:

$$s^2_{\varepsilon l} = \frac{1}{n-1}\sum_{j=1}^{n} \left(y_{jl} - \bar{y}_l \right)^2 \quad . \tag{130}$$

Обычно при проведении ДА считается, что точность измерения отклика не меняется в разных опытах. Поэтому дисперсии $s^2_{\varepsilon l}$ должны являться оценкой одной и той же генеральной дисперсии σ^2_{ε} . Проверить это можно, сравнив значимость различия максимальной и минимальной из дисперсий $s^2_{\varepsilon l}$ по критерию Фишера. Различие будет незначимым при выполнении неравенства

$$\frac{s^2_{\varepsilon \max}}{s^2_{\varepsilon \min}} \le F\left(n-1, n-1, q\right) \quad , \tag{131}$$

где $F\left(n-1, n-1, q\right)$ - квантиль распределения Фишера для числа степеней свободы сравниваемых дисперсий $l_1 = n-1$ и $l_2 = n-1$ и выбранного уровня значимости *q*. Квантиль распределения Фишера находится из таблиц по математической статистике или вычисляется с помощью функции scipy.stats.f.ppf(q, l_1 , l_2).

Если неравенство выполняется, то и все остальные дисперсии также различаются незначимо и их можно усреднить, вычислив, таким образом, оценку дисперсии ошибки

$$s_\varepsilon^2 = \frac{1}{k}\sum_{l=1}^{k} s_{\varepsilon l}^2 \quad . \tag{132}$$

Влияние фактора А можно оценить с помощью дисперсии

$$s_A^2 = \frac{n}{k-1}\sum_{l=1}^{k} (\bar{y}_l - \bar{y})^2 \quad , \tag{133}$$

где $\bar{y}$ - среднее всех наблюдений.

Для проверки значимости влияния фактора А дисперсию s_A^2 необходимо сравнить по критерию Фишера с дисперсией ошибки s_ε^2 :

$$\frac{s_A^2}{s_\varepsilon^2} \le F(k-1, k(n-1), q) \quad . \tag{134}$$

Если неравенство (134) не выполняется, то нуль-гипотеза о незначимости различия дисперсий s_A^2 и s_ε^2 отвергается, и влияние фактора А следует признать существенным.

Следует иметь в виду, что выводы, полученные с помощью дисперсионного анализа, относятся только к данному экспериментальному материалу при данной его систематизации. Поэтому, например, при изменении диапазона варьирования изучаемого фактора, или основной (базовой) его точки оценка влияния последнего может измениться.

В библиотеке scipy.stats для проведения однофакторного дисперсионного анализа имеется функция stats.f_oneway. Ниже приведен пример использования данной функции.

```
import numpy as np
```

```
from scipy import stats
np.random.seed(24)
# Создаем данные для трех групп
mu=[5,7,8]
sig=[2,2,2]
N=[15,15,15]
group=[]
for i in range(3):
    group.append(np.random.normal(mu[i], sig[i], N[i]))
# Выполнение ANOVA
F_statistic, p_value = stats.f_oneway(*group)
# Результаты
print("F-статистика:", F_statistic)
print("p-значение:", p_value)
if p_value < 0.05:
    print("Существует статистически значимая разница между группами.")
else:
    print("Нет статистически значимой разницы между группами.")
```

Результат расчета:

```
F-статистика: 13.256907141075034
p-значение: 3.4426591788341616e-05
Существует статистически значимая разница между группами.
```

Заметим, что данная программа аналогична программе сравнения нескольких выборок с использованием непараметрического теста Краскала-Уоллиса, приведенной в разделе 3.1.3.2. Отличие состоит лишь в функции для вычисления статистики, а также в том, что в случае ДА исходные выборки имеют одинаковый объем и равные стандартные отклонения. Поэтому если на практике необходимые требования для выполнения ДА (нормальное распределение в каждой группе, равенство дисперсий и независимость наблюдений) не выполняются, то адекватной альтернативой данного метода будет метод Краскала-Уоллиса.

3.2.2. Двухфакторный дисперсионный анализ

Пусть изучается влияние двух одновременно действующих факторов A и B. Фактор A варьируется на *k* уровнях, фактор B - на *m* уровнях. На каждом из сочетаний уровней проводится *n* параллельных измерений. Представим результаты наблюдений в виде таблицы (табл. 4).

Каждая ячейка таблицы отвечает определенному сочетанию уровней факторов A и B. Отклики внутри ячейки соответствуют параллельным измерениям. Общее количество опытов, включая параллельные измерения на каждом из сочетаний уровней факторов, составляет $N = kmn$. Рассеивание отклика во всех опытах можно характеризовать общей выборочной дисперсией

$$s_o^2 = \frac{1}{N-1} \sum_{i,j,q} \left(y_{ijq} - \bar{y} \right)^2 \quad , \tag{135}$$

где y_{ijq} - значение отклика на *i*-м уровне фактора A, *j*-м уровне фактора B, *q*-м параллельном измерении; $\bar{y} = \frac{1}{N} \sum_{i,j,q} y_{ijq}$ - среднее всех *N* опытов.

Эта дисперсия возникает в результате влияния фактора A, фактора B, их взаимодействия AB и фактора случайности. Задача анализа состоит в том, чтобы разделить общую выборочную дисперсию на составляющие, отвечающие вкладам факторов A (s_A^2) и B (s_B^2), их взаимодействию AB (s_{AB}^2) и вкладу фактора случайности (ошибки измерения s_ε^2)

$$s_o^2 = s_A^2 + s_B^2 + s_{AB}^2 + s_\varepsilon^2 \quad . \tag{136}$$

Таблица 4. Схема проведения опытов при двухфакторном ДА

	a_1	a_2	…	a_j	…	a_k	Средние по строкам
b_1	$y_{111}, y_{112}, \ldots y_{11n}$	$y_{121}, y_{122}, \ldots y_{12n}$	…	$y_{1j1}, y_{1j2}, \ldots y_{1jn}$	…	$y_{1k1}, y_{1k2}, \ldots y_{1kn}$	$\bar{y}_{b1}$
b_2	$y_{211}, y_{212}, \ldots y_{21n}$	$y_{221}, y_{222}, \ldots y_{22n}$	…	$y_{2j1}, y_{2j2}, \ldots y_{2jn}$	…	$y_{2k1}, y_{2k2}, \ldots y_{2kn}$	$\bar{y}_{b2}$
…	…	…	…	…	…	…	…
b_i	$y_{i11}, y_{i12}, \ldots y_{i1n}$		…	$y_{ij1}, y_{ij2}, \ldots y_{ijn}$	…	$y_{ik1}, y_{ik2}, \ldots y_{ikn}$	$\bar{y}_{bi}$
…	…	…	…	…	…	…	…
b_m	$y_{m11}, y_{m12}, \ldots y_{m1n}$	$y_{m21}, y_{m22}, \ldots y_{m2n}$	…	$y_{mj1}, y_{mj2}, \ldots y_{mjn}$	…	$y_{mk1}, y_{mk2}, \ldots y_{mkn}$	$\bar{y}_{bm}$
Средние по столбцам	$\bar{y}_{a1}$	$\bar{y}_{a2}$	…	$\bar{y}_{aj}$	…	$\bar{y}_{ak}$	

Дисперсию ошибки измерения можно оценить на основании параллельных опытов. Для этого необходимо вычислить дисперсии воспроизводимости на каждом из сочетаний уровней факторов

$$s_{ij}^2=\frac{1}{n-1}\sum_{q=1}^{n}\left(y_{ijq}-\bar{y}_{ij}\right)^2 \quad , \tag{137}$$

где $\bar{y}_{ij}=\frac{1}{n}\sum_{q} y_{ijq}$ - среднее значение отклика на каждом из сочетаний уровней факторов.

При проведении анализа предполагается, что точность измерения отклика одинакова во всех опытах, что можно проверить, сравнив по критерию Фишера наибольшую и наименьшую из дисперсий s_{ij}^2 . Дисперсии считаются однородными, если выполняется неравенство

$$\frac{s_{\max}^2}{s_{\min}^2}\le F\left(n-1,n-1,q\right) \quad , \tag{138}$$

где $s_{\max}^2$ и $s_{\min}^2$ - соответственно максимальная и минимальная дисперсии воспроизводимости.

Если дисперсии различаются незначимо (неравенство (138) выполняется), то их можно усреднить, вычислив таким образом дисперсию ошибки

$$s_{\varepsilon}^2=\frac{1}{km}\sum_{i,\,j} s_{ij}^2 \quad . \tag{139}$$

Уровни фактора А изменяются в столбцах (см. табл. 4). Для того чтобы избавиться от вклада фактора В и оценить только влияние А, необходимо усреднить значения откликов в каждом столбце. Поэтому дисперсия фактора А представляет собой рассеивание значений отклика, усредненных по столбцам, относительно общего среднего всех измерений. Она может быть вычислена по следующей формуле:

$$s_A^2=\frac{nm}{k-1}\sum_{j=1}^{k}\left(\bar{y}_{aj}-\bar{y}\right)^2 \quad , \tag{140}$$

где $\bar{y}_{aj}=\frac{1}{nm}\sum_{i,q} y_{ijq}$ - среднее значение отклика в каждом столбце.

Дисперсия фактора B характеризует рассеивание средних по строкам, так как различные уровни фактора B соответствуют строкам таблицы откликов. Дисперсия фактора B вычисляется по формуле

$$s_B^2=\frac{nk}{m-1}\sum_{i=1}^{k}\left(\bar{y}_{bi}-\bar{y}\right)^2 \quad , \tag{141}$$

где $\bar{y}_{bi}=\frac{1}{nk}\sum_{j,q} y_{ijq}$ - среднее значение отклика в каждой строке.

Дисперсия взаимодействия факторов A и B характеризует рассевание откликов относительно средних по столбцам и по строкам (рассеивание между сериями). Она вычисляется по формуле

$$s_{AB}^2=\frac{n}{(m-1)(k-1)}\sum_{i,j}\left(\bar{y}_{ij}-\bar{y}_{aj}-\bar{y}_{bi}+\bar{y}\right)^2 \quad . \tag{142}$$

Для того чтобы оценить значимость (существенность) влияния каждого из факторов (A, B и взаимодействия AB), необходимо сравнить оценки их дисперсий с оценкой дисперсии ошибки по критерию Фишера. Влияние фактора считается значимым, если оценка его дисперсии значимо превышает дисперсию ошибки, т. е. для него выполняется соответствующее неравенство

$$\frac{s_A^2}{s_\varepsilon^2} > F\left(k-1, km\left(n-1\right), q\right) \tag{143}$$

$$\frac{s_B^2}{s_\varepsilon^2} > F\left(m-1, km\left(n-1\right), q\right) \quad (144)$$

$$\frac{s_{AB}^2}{s_\varepsilon^2} > F\left(\left(k-1\right)\left(m-1\right), km\left(n-1\right), q\right) \quad . \quad (145)$$

В библиотеке scipy.stats отсутствует функция для проведения много факторного ДА. Хотя функция рассмотренная в предыдущем разделе функция scipy.stats.f_oneway изначально предназначена для однофакторного ДА, ее можно использовать и для многофакторного анализа. Для этого нужно передать ей массивы данных для каждой комбинации факторов. Однако для более продвинутого анализа, включая вычисление сумм квадратов, средних квадратов и построение таблицы реультатов ДА, лучше использовать специализированные библиотеки, такие как StatsModels. Ниже приведен текст программы для выполнения двухфакторного ДА с использованием функции anova_lm данной библиотеки.

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.formula.api import ols
# Установка случайного зерна для воспроизводимости
np.random.seed(123)
# Определение уровней факторов
factor_a = ['A1', 'A2', 'A3']
factor_b = ['B1', 'B2', 'B3']
# Генерация данных для каждой комбинации факторов с разными математическими ожиданиями
data = []
means = {
    ('A1', 'B1'): 5,
```

```
    ('A1', 'B2'): 6,
    ('A1', 'B3'): 7,
    ('A2', 'B1'): 6,
    ('A2', 'B2'): 7,
    ('A2', 'B3'): 8,
    ('A3', 'B1'): 7,
    ('A3', 'B2'): 8,
    ('A3', 'B3'): 9,
}
for a in factor_a:
    for b in factor_b:
            # Генерируем 10 случайных значений для каждой комбинации с разными математическими ожиданиями
        mean = means[(a, b)]
        values = np.random.normal(loc=mean, scale=1, size=10)
        data.extend([(a, b, value) for value in values])
# Создание DataFrame
df = pd.DataFrame(data, columns=['factor_a', 'factor_b', 'values'])
# Формула для двухфакторного ДА
formula = 'values ~ C(factor_a) + C(factor_b) + C(factor_a):C(factor_b)'
# Построение модели
model = ols(formula, df).fit()
anova_table = sm.stats.anova_lm(model, typ=2)
# Вывод результатов
print(anova_table)
```

В данной программе кроме библиотеки StatsModels используется библиотека Pandas для создания таблицы исходных данных в форме объекта DataFrame. Для задания математических ожиданий при каждом сочетании уровней факторов создается словарь means, ключами которого являются кортежи из двух текстовых переменных, обозначающих уровни факторов A и B. для каждого ключа задается некоторое (произвольное) значение математического ожидания. Далее

используем вложенный цикл для перебора всех комбинаций уровней факторов A и B. На каждой итерации во вложенном цикле:

1) Получаем соответствующее математическое ожидание из словаря means.

2) Генерируем 10 случайных значений с помощью np.random.normal(), используя полученное математическое ожидание и стандартное отклонение, равное 1.

3) Создаем список кортежей, состоящих из уровней факторов A и B и сгенерированного значения и добавляем созданные кортежи в список data.

После окончания цикла создаем DataFrame df из списка data, используя колонки «factor_a», «factor_b» и «values». В результате мы получаем таблицу, в которой первые два столбца обозначают уровни факторов A и B, а в третьем столбце содержатся данные, отвечающие каждому сочетанию уровней.

Методика работы с функцией anova_lm предусматривает следующие шаги:

1) Создание формулы ДА. Определяем текстовую переменную formula, в которую записываем модель проводимого анализа. В данном случае используется формула

'values ~ C(factor_a) + C(factor_b) + C(factor_a):C(factor_b)',

согласно которой данные включают сумму эффектов фактора A (C(factor_a)), фактора B (C(factor_b)) и их взаимодействия (C(factor_a):C(factor_b)).

2) Построение модели. Используя функцию ols (Ordinary Least Squares) из statsmodels.formula.api, мы строим модель ДА, передавая ей формулу formula и DataFrame df. Метод fit() применяется для оценки параметров модели по алгоритму наименьших квадратов.

model = ols(formula, df).fit()

3) Вычисление таблицы ДА. Функция anova_lm из statsmodels.api используется для вычисления таблицы ДА. Аргумент typ=2 указывает на использование суммы квадратов типа II (рекомендуется для несбалансированных данных).

В последней инструкции программы осуществляется вывод результатов анализа. Мы выводим таблицу ДА, содержащую суммы квадратов, степени свободы, F-статистики и p-значения для каждого источника вариации (факторы A и B, их взаимодействие и остаточная вариация).

Результат работы программы:

	sum_sq	df	F	PR(>F)
C(factor_a)	53.318659	2.0	20.231062	7.478045e-08
C(factor_b)	90.032166	2.0	34.161517	1.743844e-11
C(factor_a):C(factor_b)	7.381048	4.0	1.400321	2.412996e-01
Residual	106.737142	81.0	NaN	NaN

Разберем результат.

Структура таблицы ДА.

1. Source: Это источники вариации в данных:

- C(factor_a): Вариация, объясненная фактором A.

- C(factor_b): Вариация, объясненная фактором B.

- C(factor_a):C(factor_b): Вариация, объясненная взаимодействием факторов A и B.

- Residual: Остаточная вариация, которая не объясняется моделируемыми факторами.

Параметры таблицы.

2. sum_sq: Сумма квадратов для каждого источника вариации.

- C(factor_a): 53.318659 — это сумма квадратов вариации, объясненной фактором A. Это означает, что фактор A вносит значительный вклад в общую вариацию данных.

- C(factor_b): 90.032166 — это сумма квадратов вариации, объясненной фактором B. Это также указывает на значительный вклад фактора B.

- C(factor_a):C(factor_b): 7.381048 — это сумма квадратов вариации, объясненной взаимодействием факторов A и B. Она значительно меньше, чем суммы квадратов для факторов A и B, что говорит о том, что взаимодействие не столь существенно.

- Residual: 106.737142 — это остаточная вариация, которая не объясняется факторами A и B.

3. df: Степени свободы для каждого источника вариации.

- Для C(factor_a) и C(factor_b) степени свободы равны количеству уровней фактора минус один (3 - 1 = 2).

- Для взаимодействия C(factor_a):C(factor_b) степени свободы равны произведению степеней свободы каждого фактора (2 * 2 = 4).

- Остаточные степени свободы рассчитываются как общее количество наблюдений минус количество уровней факторов (90 - 9 = 81).

4. F: F-статистика для каждого источника вариации.

- C(factor_a): 20.231062 — это высокая F-статистика, что указывает на то, что вариация, объясненная фактором A, значительно превышает остаточную вариацию.

- C(factor_b): 34.161517 — это также высокая F-статистика, указывающая на значительный вклад фактора B.

- C(factor_a):C(factor_b): 1.400321 — это значение F для взаимодействия факторов A и B, которое значительно ниже, что говорит о том, что взаимодействие не является значимым.

5. PR(>F): p-value для каждой F-статистики.

- C(factor_a): 7.478045e-08 — это очень низкое p-значение, что указывает на наличие статистически значимых различий между уровнями фактора A. Это значит, что фактор A влияет на значения отклика.

- C(factor_b): 1.743844e-11 — это также очень низкое p-значение, указывающее на наличие статистически значимых различий между уровнями фактора B.

- C(factor_a):C(factor_b): 2.412996e-01 — это p-значение значительно выше 0.05, что указывает на отсутствие статистически значимого взаимодействия между факторами A и B.

На основании полученных результатов можно сделать следующие выводы:

- Статистически значимые эффекты: Оба фактора A и B имеют статистически значимые эффекты на значения отклика, так как их p-значения значительно меньше 0.05.

- Не значимое взаимодействие: Взаимодействие между факторами A и B не является статистически значимым, поскольку p-значение для взаимодействия больше 0.05.

- Практическое значение: Это означает, что изменения в значениях отклика можно объяснить изменениями в факторах A и B, но взаимодействие между ними не оказывает значительного влияния на отклик, то есть их влияние можно рассматривать независимо.

3.2.3. Планирование эксперимента при дисперсионном анализе

Выше рассмотрен двухфакторный дисперсионный анализ, включающий все возможные сочетания факторов. Такой способ постановки эксперимента называется полным факторным экспериментом (ПФЭ). В некоторых случаях исходя из смысла исследуемого объекта заранее известно, что эффект взаимодействия между факторами отсутствует или заведомо мал по сравнению с влиянием самих факторов. В такой ситуации количество одновременно исследуемых факторов может быть увеличено без увеличения числа опытов путем введения в план эксперимента новых факторов. Такой способ постановки эксперимента (план эксперимента) называется дробным факторным экспериментом (ДФЭ).

При постановке ДФЭ важно так спланировать эксперимент, чтобы терялось наименьшее количество информации. В методе дисперсионного анализа это достигается путем введения в план эксперимента новых факторов по схеме так называемых ортогональных латинских квадратов. Латинский квадрат – это квадратная таблица, составленная из элементов таким образом, что каждый элемент повторяется в каждой строке и в каждом столбце только один раз. Например, латинский квадрат 3x3 из элементов A, B, C:

A	B	C
B	C	A
C	B	A

Наиболее простой способ построения латинских квадратов – это одношаговая циклическая перестановка. Так, при исследовании трех факторов (A, B, C) на двух уровнях третий фактор может быть введен в план по схеме латинского квадрата:

B	A	
	a_1	a_2
b_1	c_1	c_2
b_2	c_2	c_1

Как видно, при изменении двух факторов A и B в каждом опыте изменяется одновременно и третий фактор C, причем так, что его уровни не повторяются ни в одной строке, ни в одном столбце. Схема постановки опытов (матрица планирования эксперимента) для такого плана имеет вид:

Номер опыта	Уровни факторов			Отклики
	A	B	C	
1	a_1	b_1	c_1	y_1
2	a_1	b_2	c_2	y_2
3	a_2	b_1	c_2	y_3
4	a_2	b_2	c_1	y_4

Планирование эксперимента по схеме латинского квадрата позволяет одновременно изучить влияние трех факторов. Следует иметь в виду, что при введении в план третьего фактора по схеме латинского квадрата его эффект оказывается смешанным с эффектом взаимодействия первых двух

факторов, поэтому в качестве этих основных факторов следует выбирать те, взаимодействие между которыми незначимо.

Дисперсионный анализ латинского квадрата без повторных наблюдений производится по следующему алгоритму:

1) Вычислить суммы значений откликов по уровням фактора A (итоги по столбцам A_i), по уровням фактора B (итоги по строкам B_j) и по уровням фактора C (итоги по латинским буквам C_q). В последнем случае надо просуммировать значения отклика в тех группах опытов, где фактор C принимает одно и то же значение,

$$A_1 = y_1 + y_2 \qquad A_2 = y_3 + y_4 \quad ,$$

$$B_1 = y_1 + y_3 \qquad B_2 = y_2 + y_4 \quad ,$$

$$C_1 = y_1 + y_4 \qquad C_2 = y_2 + y_3 \quad ,$$

2) сумму квадратов всех наблюдений:

$$SS_1 = \sum_{i=1}^{n} \sum_{j=1}^{n} y_{ij}^2 = y_1^2 + y_2^2 + y_3^2 + y_4^2 \quad ,$$

3) сумму квадратов итогов по столбцам, деленную на число наблюдений в столбце:

$$SS_2 = \frac{1}{n} \sum_{i=1}^{n} A_i^2 = \frac{1}{2} \left(A_1^2 + A_2^2 \right) \quad ,$$

4) сумму квадратов итогов по строкам, деленную на число наблюдений в строке:

$$SS_3 = \frac{1}{n} \sum_{j=1}^{n} B_j^2 = \frac{1}{2} \left(B_1^2 + B_2^2 \right) \quad ,$$

5) сумму квадратов итогов по латинским буквам, деленную на число наблюдений, соответствующих каждой латинской букве:

$$SS_4 = \frac{1}{n}\sum_{q=1}^{n} C_q^2 = \frac{1}{2}\left(C_1^2 + C_2^2\right) \quad ,$$

6) квадрат общего итога, деленный на число всех наблюдений (корректирующий член):

$$SS_5 = \frac{1}{n^2}\left(\sum_{i=1}^{n} A_i\right)^2 = \frac{1}{4}\left(y_1 + y_2 + y_3 + y_4\right)^2 \quad ,$$

7) сумму квадратов для столбца:

$$SS_A = SS_2 - SS_5 \quad ,$$

8) сумму квадратов для строки:

$$SS_B = SS_3 - SS_5 \quad ,$$

9) сумму квадратов для латинской буквы:

$$SS_C = SS_4 - SS_5 \quad ,$$

10) общую сумму квадратов, равную разнице между суммой квадратов всех наблюдений и корректирующим членом:

$$SS_{общ} = SS_1 - SS_5 \quad ,$$

11) остаточную сумму квадратов:

$$SS_{ост} = SS_{общ} - SS_A - SS_B - SS_C \quad .$$

Остаточная сумма квадратов складывается из дисперсии, обусловленной ошибкой опыта, и дисперсии, обусловленной взаимодействиями факторов, если таковые имеются;

12) дисперсию фактора А:

$$s_A^2 = \frac{SS_A}{n-1} \quad ,$$

13) дисперсию фактора B:

$$s_B^2 = \frac{SS_B}{n-1} \quad ,$$

14) дисперсию фактора C:

$$s_C^2 = \frac{SS_C}{n-1} \quad ,$$

12) остаточную дисперсию:

$$s_{ост}^2 = \frac{SS_{ост}}{(n-1)(n-2)} \quad .$$

Значимость влияния факторов проверяют по критерию Фишера. Влияние фактора считается значимым, если для него выполняется соответствующее неравенство:

$$\frac{s_A^2}{s_{ост}^2} > F(n-1,(n-1)(n-2),q)$$

$$\frac{s_B^2}{s_{ост}^2} > F(n-1,(n-1)(n-2),q)$$

$$\frac{s_C^2}{s_{ост}^2} > F(n-1,(n-1)(n-2),q)$$

Можно ввести в план эксперимента и дополнительные факторы, т. е. наложить несколько латинских квадратов один на другой. Такие планы называются греко-латинскими квадратами (если дополнительно вводится в план два фактора) и гипер-греко-латинскими квадратами (если в план дополнительно вводится более двух факторов). Наложение одного латинского квадрата на другой следует производить ортогонально, т. е. так, чтобы каждая латинская буква одного квадрата была связана с каждой латинской буквой другого квадрата один и только один раз.

Пример греко-латинского квадрата 3x3:

B	A		
	a_1	a_2	a_3
b_1	c_1 d_1	c_2 d_2	c_3 d_3
b_2	c_2 d_3	c_3 d_1	c_1 d_2
b_3	c_3 d_2	c_1 d_3	c_2 d_1

В данном примере факторы A и B являются основными, а факторы C и D введены в план эксперимента в виде двух ортогональных латинских квадратов. Оба они получены путем одношаговой циклической перестановки элементов. Ортогональность достигается тем, что для фактора C перестановка элементов осуществляется слева направо, а для фактора D справа налево, таким образом, каждое сочетание уровней факторов встречается в плане эксперимента только один раз. При возрастании числа факторов задача построения ортогональных латинских квадратов резко усложняется и в общем случае в комбинаторной математике до конца не решена. Максимальное количество факторов, вводимых по схеме латинского квадрата, при n уровнях основных факторов составляет $n+1$. Такой план называется насыщенным, поскольку число степеней свободы остаточной суммы квадратов, определяемое по формуле $f=(n-1)(n-k+1)$, где k – число изучаемых факторов, равно нулю. Насыщенные планы обычно применяют на начальных стадиях исследования, когда приходится проводить перебор качественных факторов, с тем, чтобы выделить наиболее перспективные комбинации и отсеять неприемлемые. Ниже приведен пример насыщенного плана ортогонального гипер-греко-латинского квадрата для $n=5$.

A	B				
	0	1	2	3	4
0	C=0 D=0 E=0 F=0	C=1 D=1 E=1 F=1	C=2 D=2 E=2 F=2	C=3 D=3 E=3 F=3	C=4 D=4 E=4 F=4
1	C=1 D=2 E=3 F=4	C=2 D=3 E=4 F=0	C=3 D=4 E=0 F=1	C=4 D=0 E=1 F=2	C=0 D=1 E=2 F=3
2	C=2 D=4 E=1 F=3	C=3 D=0 E=2 F=4	C=4 D=1 E=3 F=0	C=0 D=2 E=4 F=1	C=1 D=3 E=0 F=2
3	C=3 D=1 E=4 F=2	C=4 D=2 E=0 F=3	C=0 D=3 E=1 F=4	C=1 D=4 E=2 F=0	C=2 D=0 E=3 F=1
4	C=4 D=3 E=2 F=1	C=0 D=4 E=3 F=2	C=1 D=0 E=4 F=3	C=2 D=1 E=0 F=4	C=3 D=2 E=1 F=0

Дисперсионный анализ греко-латинских квадратов проводится так же, как и латинских. Число степеней свободы факторов одинаково и равно *n*-1, а число степеней свободы остаточной суммы равно $f=(n-1)(n-k+1)$, где *k* – число изучаемых факторов.

3.3. Линейный регрессионный анализ

Зависимость между двумя случайными величинами *X* и *У* полностью задана, если известна их совместная функция распределения $f(x,y)$. На практике, однако, часто бывает удобнее пользоваться не функцией распределения, а зависимостью математического ожидания одной из случайных величин от другой, например $m_y(x)$. Такая зависимость называется *регрессией*. Уравнение, выражающее эту зависимость, называется *уравнением регрессии*. Широкая практическая применимость уравнений регрессии связана с разделением переменных математических моделей на входные и выходные, или зависимые и независимые. Исследователя, как правило, интересует количественное описание, в виде математического выражения, зависимости выходных переменных от входных. В условиях случайных погрешностей измерения выходных переменных такое математическое выражение и будет уравнением регрессии.

Различают однофакторный регрессионный анализ, когда входная, или независимая, переменная только одна, и многофакторный, когда выходная величина является функцией нескольких входных переменных.

Задача регрессионного анализа состоит в том, чтобы на основании ограниченного объема экспериментальных данных (выборки) найти приближенное уравнение регрессии и оценить возникающую при этом ошибку.

3.3.1. Уравнения регрессии

Вид уравнения регрессии может быть различным, в частности, уравнение может отражать некоторую физическую или физико-химическую закономерность, лежащую в основе изучаемого явления. Однако найти обоснованную физическую модель, описывающую рассматриваемый процесс, удается далеко не всегда. Иногда даже имеющаяся такая модель может оказаться слишком

сложной и неудобной для практического использования. В этих случаях уравнение регрессии выбирают в виде полинома. Ниже приводится вид уравнений полиномиальной регрессии.

Для однофакторной модели

$$y = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \ldots \quad , \tag{146}$$

где y - исследуемый отклик, x - независимая переменная (фактор), $b_0, b_1, b_2, \ldots$ - искомые коэффициенты уравнения регрессии (параметры модели).

Для многофакторной модели, имеющей *k* независимых переменных:

$$y = b_0 + \sum_{i=1}^{k} b_i x_i + \sum_{i=1}^{k} b_{ii} x_i^2 + \sum_{i,j;i<j}^{k} b_{ij} x_i x_j + \ldots \quad , \tag{147}$$

где $x_i, i = 1..k$ - независимые переменные, b_0, b_i, b_{ii}, b_{ij} - коэффициенты уравнения регрессии.

В многофакторных моделях коэффициенты уравнения регрессии обычно называют следующим образом: b_0 - свободный член, b_i - линейные члены, b_{ii} - члены второго порядка (квадратичные эффекты), b_{ij} - эффекты парного взаимодействия и т. д.

Возможность представления математических моделей в виде полиномов вида (146) и (147) обосновывается тем, что для большинства процессов модели описываются непрерывными функциями входных параметров и по теореме Тейлора могут быть разложены в степенной ряд. Удобство представления математических моделей в виде полиномов связано еще и с тем, что, несмотря на то, что независимая переменная y является нелинейной функцией аргументов x_i, искомые параметры модели (коэффициенты уравнения регрессии $b_0, b_i \ldots$) входят в уравнение регрессии в первой степени. Такая модель называется линейной по параметрам. Именно поэтому настоящий раздел называется «Линейный регрессионный анализ» в отличие от

«Нелинейного регрессионного анализа», при котором регрессионная модель не является линейной по параметрам. Ниже будет показано, что для моделей, линейных по параметрам, коэффициенты регрессии вычисляются с помощью достаточно простых формул. В случае моделей, нелинейных по параметрам, численная процедура оценки коэффициентов регрессии существенно усложняется, в связи с чем данный метод будет рассмотрен в отдельном разделе, после рассмотрения методов решения задачи оптимизации.

3.3.2. Метод наименьших квадратов

Оценки коэффициентов уравнения регрессии будут зависеть от того, каким способом определить приближение экспериментальных данных уравнением регрессии. Чаще всего для такого приближения используется метод наименьших квадратов (МНК). Суть его состоит в том, что оценки коэффициентов вычисляются из условия минимума функции $\Phi(b_0, b_i, b_{ii}\ldots)$ следующего вида:

$$\Phi(b_0, b_i, b_{ii}\ldots) = \sum_{j=1}^{N} (y_j - \hat{y}_j)^2 \to \min \quad , \tag{148}$$

где y_j - экспериментальное значение отклика в j -м опыте, N - количество опытов (объем выборки), $\hat{y}_j$ - значение отклика, рассчитанное по уравнению регрессии для соответствующих условиям данного опыта значений независимых переменных.

То есть коэффициенты уравнения регрессии вычисляются таким образом, чтобы минимизировать сумму квадратов отклонений расчетных и экспериментальных значений отклика во всех опытах.

Рассмотрим метод наименьших квадратов на примере вычисления коэффициентов многофакторного уравнения линейной регрессии от k независимых переменных.

$$y = b_0 + \sum_{i=1}^{k} b_i x_i \quad . \tag{149}$$

Для упрощения дальнейших вычислений удобно ввести дополнительную переменную x_0 , значение которой положить тождественно равным единице. Тогда уравнение (149) запишется более просто:

$$y=\sum_{i=0}^{k} b_i x_i \quad . \tag{150}$$

Пусть в результате N экспериментов получена выборка экспериментальных значений отклика $y_j, j=1..N$. В каждом из этих опытов независимые переменные принимали значения $x_{ji}, j=1..N, i=1..k$. Для уравнения регрессии (150) функция (148) будет иметь вид

$$\Phi(b_i)=\sum_{j=1}^{N}\left(y_j-\sum_{i=0}^{k} b_i x_{ji}\right)^2 \quad . \tag{151}$$

Обращаем внимание, что аргументами функции Φ являются именно коэффициенты уравнения регрессии b_i , а не независимые переменные модели x_i . Значения y_j и x_{ji} являются просто числами, полученными на основании эксперимента.

Как известно из анализа, необходимым условием экстремума функции нескольких переменных является равенство нулю ее частных производных по каждому из аргументов. Отсюда получаем условие минимума функции (151):

$$\begin{aligned}
&\frac{\partial \Phi}{\partial b_0}=0\\
&\frac{\partial \Phi}{\partial b_1}=0\\
&\dots\\
&\frac{\partial \Phi}{\partial b_i}=0\\
&\dots\\
&\frac{\partial \Phi}{\partial b_k}=0
\end{aligned} \qquad (152)$$

Система (152) называется системой нормальных уравнений МНК. Выполнив дифференцирование, получим

$$\begin{aligned}
&\sum_{j=1}^{N} y_j x_{j0} - \sum_{j=1}^{N} x_{j0} \sum_{i=0}^{k} b_i x_{ji} = 0\\
&\sum_{j=1}^{N} y_j x_{j1} - \sum_{j=1}^{N} x_{j1} \sum_{i=0}^{k} b_i x_{ji} = 0\\
&\dots\\
&\sum_{j=1}^{N} y_j x_{ji} - \sum_{j=1}^{N} x_{ji} \sum_{i=0}^{k} b_i x_{ji} = 0\\
&\dots\\
&\sum_{j=1}^{N} y_j x_{jk} - \sum_{j=1}^{N} x_{jk} \sum_{i=0}^{k} b_i x_{ji} = 0
\end{aligned} \qquad (153)$$

Как видно, система (153) представляет собой линейную систему из $k+1$ уравнений относительно $k+1$ неизвестных $b_0, b_1, .. b_i, .. b_k$. Решая ее, находят оценки коэффициентов уравнений регрессии.

Линейный регрессионный анализ удобно представлять в матричной форме. Обозначим двумерный массив значений независимых переменных в N опытах в виде матрицы X размером $N\cdot(k+1)$, столбцы которой соответствуют значениям независимых переменных:

$$X=\begin{pmatrix} x_{10} & x_{11} & \dots & x_{1k} \\ x_{20} & x_{21} & \dots & x_{1k} \\ \dots & \dots & \dots & \dots \\ x_{N0} & x_{N1} & \dots & x_{Nk} \end{pmatrix} \qquad (154)$$

Искомые значения оценок коэффициентов уравнения регрессии можно представить в виде вектора

$$\hat{b}=\begin{pmatrix} \hat{b}_0 \\ \hat{b}_1 \\ \dots \\ \hat{b}_k \end{pmatrix} \qquad (155)$$

При известных значениях коэффициентов b значения отклика в каждом из N опытов y_p можно рассчитать с использованием следующего матричного выражения:

$$y_p=X\,b \quad . \qquad (156)$$

В силу наличия случайных погрешностей экспериментальные значения отклика $y_э$ будут отличаться от рассчитанных по уравнению (156) на некоторые величины, которые можно представить в виде вектора "невязок" $\delta=y_p-y_э$. В соответствии с идеей метода наименьших квадратов коэффициенты уравнения регрессии (150) следует выбрать таким образом, чтобы обеспечить минимум нормы вектора δ :

$$(\delta, \delta) \rightarrow \min \quad , \tag{157}$$

что в развернутой форме эквивалентно минимуму выражения (151). Можно заметить, что при выбранной системе обозначений система нормальных уравнений (153) в матричной форме запишется следующим образом:

$$(X^T X)\hat{b} = X^T y_э \quad , \tag{158}$$

где X^T - транспонированная матрица X .

Умножив слева обе части выражения (158) на матрицу, обратную к $X^T X$, получаем окончательно в матричной форме выражение для расчета коэффициентов уравнения регрессии:

$$\hat{b} = (X^T X)^{-1} X^T y_э \quad . \tag{159}$$

Помимо того, что выражение (159) является компактной формой записи решения системы нормальных уравнений, оно удобно для ее теоретического анализа и фактически представляет собой алгоритм вычисления коэффициентов уравнения регрессии на компьютере с использованием программ, в которых реализованы матричные операции.

Ниже приведен текст программы, иллюстрирующей вычисление оценок коэффициентов регрессии по формуле (159).

```
import numpy as np
from numpy.linalg import inv
# Функция для моделирования наблюдений со случайной ошибкой
def get_yerr(y,sigma,N):
   err=np.random.normal(0, sigma, N)
   return(y+err)
N=15 # Объем выборки
k=3  # Количество коэффициентов, не считая b_0
sigma=0.1 # Стандартное отклонение ошибки наблюдения
X=np.random.rand(N,k)*10 # Генерируем матрицу независимых переменных
```

```
X=np.hstack([np.ones((N,1)),X]) # Добавляем столбец для свободного члена
b=np.random.randint(1,10,k+1) # Генерируем истинные значения коэффициентов
y=X @ b # Вычисляем истинные значения откликов
y_err=get_yerr(y,sigma,N) # Моделируем значения откликов с ошибкой err
hat_b=inv(X.T @ X)@(X.T @ y_err) # Вычисляем оценки коэффициентов
print(f"Истинные коэффициенты регрессии: {b}")
print(f"Оценки коэффициентов регрессии: {hat_b}")
```

В данной программе значение независимых переменных генерируется матрицей X размером $N\times(k+1)$, в которой первый столбец состоит из единиц для моделирования свободного члена уравнения регрессии, а остальные генерируются случайными числами с равномерным распределением в интервале [0,10]. Значения истинных коэффициентов регрессии генерируется целыми случайными числами для удобства последующего сравнения с оценками. Функция `y=X @ b` вычисляет истинные значения коэффициентов по формуле (156). Символ «@» означает матричное произведение в numpy. Для транспонирования матрицы `X` используется символ «X.T». Функция «inv» вычисляет обратную матрицу.

Результат работы программы при N=15:

```
Истинные коэффициенты регрессии: [1 6 3 5]
Оценки коэффициентов регрессии: [0.90055869 6.01186029 3.00862396 4.99829327]
```

Результат при N=150:

```
Истинные коэффициенты регрессии: [2 7 6 1]
Оценки коэффициентов регрессии: [2.03711821 7.0006553  6.0007233  0.99022385]
```

Как видно, с увеличением объема выборки оценки коэффициентов регрессии начинают менее отличаться от истинных значений.

Мы рассмотрели вычисление коэффициентов линейного уравнения регрессии (150). Фактически же этот метод без изменения применим к любой модели, линейной по параметрам, поскольку путем соответствующей замены переменных она приводится к уравнению линейной регрессии. Так, например, уравнение однофакторной регрессии (146) путем замены переменных $x_0=1, x_1=x, x_2=x^2, x_3=x^3, \ldots$ сводится к уравнению многофакторной линейной регрессии (150). Уравнение многофакторной регрессии (147) путем введения

новых переменных вместо квадратичных членов и членов взаимодействий $x_{k+1}=x_1^2, x_{k+2}=x_2^2, \ldots x_{2k}=x_k^2, x_{2k+1}=x_1 x_2, x_{2k+2}=x_1 x_3, \ldots$ также сводится к уравнению линейной регрессии, но от большего числа независимых переменных.

3.3.3. Статистический анализ уравнения регрессии

После того как рассчитаны коэффициенты уравнения регрессии, необходимо провести его статистический анализ. Анализ включает проверку адекватности уравнения и проверку значимости коэффициентов.

Наиболее важной представляется проверка адекватности уравнения, поскольку она определяет допустимость его практического использования. Уравнение считается адекватным, или соответствующим экспериментальным данным, если ошибка, возникающая вследствие неточности модели, описываемой уравнением регрессии, по отношению к истинной зависимости, описываемой экспериментальными данными, не превышает погрешности эксперимента.

Степень неточности (неадекватности) модели можно оценить с помощью дисперсии адекватности

$$s_{ад}^2=\frac{1}{N-l}\sum_{i=1}^{N}\left(y_i^э-y_i^p\right)^2 \quad , \qquad (160)$$

где N - объем выборки (число опытов), l - число определяемых на основании этой выборки коэффициентов уравнения регрессии (число связей, накладываемых на выборку), $y_i^э$ - экспериментальные значения отклика в каждом опыте, y_i^p - значения отклика, рассчитанные по уравнению регрессии. Величина, входящая в числитель формулы (160), называется остаточной суммой квадратов.

Ошибку наблюдения оценивают по результатам параллельных измерений

$$s_{ош}^2=\frac{1}{n-1}\sum_{i=1}^{n}\left(y_i-\bar{y}\right)^2 \quad , \qquad (161)$$

где n - число параллельных измерений, $\bar{y}$ - среднее значение отклика в параллельных измерениях.

Вывод об адекватности модели, описываемой уравнением регрессии, можно сделать на основании проверки значимости различия дисперсии адекватности и дисперсии ошибки, которую можно провести с помощью критерия Фишера. Различие между дисперсиями можно признать незначимым, а уравнение регрессии - соответственно адекватным эксперименту при выполнении неравенства

$$\frac{s_{ад}^2}{s_{ош}^2} \le F\left(N-l, n-1, q\right) \quad , \tag{162}$$

где $F\left(N-l, n-1, q\right)$ - квантиль распределения Фишера (критическое значение критерия Фишера) для числа степеней свободы дисперсии в числителе $v_1 = N-l$, в знаменателе - $v_2 = n-1$ и выбранного уровня значимости q .

При отсутствии параллельных опытов приближенно адекватность уравнения регрессии можно оценить по величине критерия R^2 (коэффициент достоверности аппроксимации или коэффициент детерминированности модели), вычисляемого по следующей формуле:

$$R^2 = 1 - \frac{\sum_{i=1}^{N} \left(y_i^э - y_i^p\right)^2}{\sum_{i=1}^{N} \left(y_i^э - \bar{y}\right)^2} \quad , \tag{163}$$

где $\bar{y}$ - среднее всех опытов.

Близость вычисленного значения R^2 к единице свидетельствует о хорошей адекватности уравнения.

Значимость коэффициентов уравнения регрессии проверяют по критерию Стьюдента. Коэффициент уравнения регрессии b_j считается значимым, если отношение его абсолютной величины к среднеквадратическому отклонению оценки коэффициента превышает критическое значение критерия Стьюдента

$$\frac{|b_j|}{s_{b_j}} > t(q, \nu) \quad , \tag{164}$$

где s_{b_j} - среднеквадратичное отклонение *j*-го коэффициента, $t(q, \nu)$ - критическое значение критерия Стьюдента для выбранного уровня значимости q и числа степеней свободы дисперсии воспроизводимости ν .

Среднеквадратическое отклонение коэффициента определяется по закону сложения ошибок

$$s_{b_j} = \sqrt{\sum_{i=1}^{N} \left(\frac{\partial b_j}{\partial y_i} \right)^2 s_i^2} \quad , \tag{165}$$

где s_i^2 - дисперсия воспроизводимости в *i*-м опыте.

Незначимость коэффициента свидетельствует о том, что влияние фактора, отвечающего данному коэффициенту, не существенно. Незначимые коэффициенты при желании могут быть исключены из уравнения. Однако следует иметь в виду, что в общем случае все оценки коэффициентов уравнения регрессии закоррелированы между собой. Поэтому при исключении из уравнения одного из коэффициентов остальные должны быть пересчитаны заново.

Ниже приведен вариант программы, рассмотренной в предыдущем разделе, в которой добавлен статистический анализ уравнения регрессии.

```
import numpy as np
from numpy.linalg import inv
import scipy.stats as stats
import matplotlib.pyplot as plt
# Функция для моделирования наблюдений со случайной ошибкой
```

```
def get_yerr(y,sigma,N):
    err=np.random.normal(0, sigma, N)
    S2err=np.mean(err**2)
    return(y+err,S2err)
N=15 # Объем выборки
k=3  # Количество коэффициентов, не считая b_0
X=np.random.rand(N,k)*10 # Генерируем матрицу независимых переменных
X=np.hstack([np.ones((N,1)),X]) # Добавляем столбец для свободного члена
b=np.random.randint(1,10,k+1) # Генерируем истинные значения коэффициентов
y=X @ b # Вычисляем истинные значения откликов
sigma=0.5
y_err,S2err=get_yerr(y,sigma,N)
hat_b=inv(X.T @ X)@(X.T @ y_err)
y_calc=X @ hat_b
dof = (N-k)  # Степени свободы для выборки
S2=((y_calc-y_err) @ (y_calc-y_err))/dof # Дисперсия адекватности
F_stat=S2/(S2err)
y_mean=np.mean(y_err)
R2=1-((y_calc-y_err) @ (y_calc-y_err))/((y_err-y_mean) @ (y_err-y_mean))
alpha = 0.05  # Уровень значимости
# Доверительный интервал для выборочной дисперсии
chi2_lower = stats.chi2.ppf(alpha / 2, dof)
chi2_upper = stats.chi2.ppf(1 - alpha / 2, dof)
ci_lower = (dof * S2) / chi2_upper
ci_upper = (dof * S2) / chi2_lower
f_critical = stats.f.ppf(1 - alpha, dof, N-1)
print(f"Истинные коэффициенты регрессии: {b}")
print(f"Оценки коэффициентов регрессии: {hat_b}")
print(f"Дисперсия адекватности: {S2}")
print(f"95% доверительный интервал для дисперсии: ({ci_lower}, {ci_upper})")
print(f"Дисперсия ошибки:  {S2err}")
```

```
print(f"F-статистика:  {F_stat}")
print(f"Критическое значение F: {f_critical}")
print(f"Коэффициент детерминации:  {R2}")
```

В функцию get_yerr(y,sigma,N) добавлено вычисление оценки дисперсии ошибки S2err, путем усреднения ошибок, генерируемых в выборке нормального распределения. Вычисление дисперсии адекватности уравнения регрессии S2 осуществляется по формуле (160). Границы доверительного интервала дисперсии адекватности ci_lower, ci_upper вычисляются аналогично (79). F-статистика F_stat вычисляется как отношение дисперсий S2/(S2err). Критическое значение критерия Фишера вычисляется с помощью функции квантиля распределения Фишера stats.f.ppf(1 - alpha, dof, N-1). Коэффициент детерминации рассчитывается по формуле (163).

Ниже приведен вариант вывода результатов расчета.

```
Истинные коэффициенты регрессии: [7 6 1 7]
Оценки коэффициентов регрессии: [7.26846229 5.95927584 0.90630018 7.0813355 ]
Дисперсия адекватности: 0.35430384412372184
95% доверительный интервал для дисперсии: (0.1821873983608574, 0.9654519336576143)
Дисперсия ошибки:  0.40490796937162343
F-статистика:  0.875023143341846
Критическое значение F: 2.534243252748561
Коэффициент детерминации:  0.999521143185927
```

На основании результатов расчетов можно сделать вывод что модель адекватна экспериментальным данным, поскольку с одной стороны оценка дисперсии ошибки входит в доверительный интервал дисперсии адекватности, а с другой стороны, вычисленное значение F-статистики существенно ниже критического значения квантиля распределения Фишера. Также об адекватности модели свидетельствует близость к 1 коэффициента детерминации.

Для регрессионного анализа можно использовать также библиотеку statmodels, которая предоставляет удобные инструменты для статистического анализа. Ниже приведен пример программы, которая генерирует случайные данные, выполняет линейную регрессию и оценивает значимость коэффициентов.

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
# Устанавливаем случайное начальное значение для воспроизводимости
np.random.seed(0)
# Генерируем случайные данные
N = 15  # Количество наблюдений
X1 = np.random.rand(N) * 10  # Первый фактор
X2 = np.random.rand(N) * 10  # Второй фактор
X3 = np.random.rand(N) * 10  # Третий фактор
sigma=0.5
err = np.random.normal(0, sigma, N)  # Ошибка
# Задаем истинные коэффициенты 7 6 1 7
beta_0 = 7
beta_1 = 6
beta_2 = 1
beta_3 = 7
# Генерируем зависимую переменную Y
Y = beta_0 + beta_1 * X1 + beta_2 * X2 + beta_3 * X3 + err
# Создаем DataFrame для независимых переменных
X = pd.DataFrame({'X1': X1, 'X2': X2, 'X3': X3})
# Добавляем константу для свободного члена
X = sm.add_constant(X)
# Создаем и обучаем модель линейной регрессии
model = sm.OLS(Y, X).fit()
# Выводим результаты
print(model.summary())
```

Программа выдала результат:

```
OLS Regression Results
==============================================================================
=====
```

```
Dep. Variable:                    y   R-squared:                       1.000
Model:                          OLS   Adj. R-squared:                  1.000
Method:               Least Squares   F-statistic:                  1.208e+04
Date:              Wed, 07 Aug 2024   Prob (F-statistic):            1.26e-19
Time:                      13:07:25   Log-Likelihood:                -7.2610
No. Observations:                15   AIC:                             22.52
Df Residuals:                    11   BIC:                             25.35
Df Model:                         3
Covariance Type:          nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.7845      0.452     15.023      0.000       5.790       7.779
X1             5.9884      0.053    114.009      0.000       5.873       6.104
X2             1.0018      0.037     27.205      0.000       0.921       1.083
X3             7.0163      0.048    146.578      0.000       6.911       7.122
==============================================================================
Omnibus:                        0.036   Durbin-Watson:                   2.871
Prob(Omnibus):                  0.982   Jarque-Bera (JB):                0.134
Skew:                          -0.069   Prob(JB):                        0.935
Kurtosis:                       2.557   Cond. No.                         39.0
==============================================================================
```

По результату можно сделать следующие пояснения.

1. Статистики модели:

- `R-squared` и `Adj. R-squared`: Коэффициенты детерминации, показывающие долю объясненной дисперсии в общей дисперсии зависимой переменной.

- `F-statistic` и `Prob (F-statistic)`: F-статистика и ее p-значение для проверки значимости модели в целом.

2. Коэффициенты регрессии:

- `const`: Свободный член (intercept) модели.

- `X1`, `X2`, `X3`: Коэффициенты при независимых переменных.

- `coef`: Оценки коэффициентов, полученные методом наименьших квадратов.

- `std err`: Стандартные ошибки коэффициентов.

- `t` и `P>|t|`: t-статистики и их p-значения для проверки значимости коэффициентов.

- `[0.025`, `0.975]`: 95% доверительный интервал для коэффициентов.

3. Диагностические статистики:

- `Omnibus`, `Prob(Omnibus)`: Тест на нормальность распределения остатков.

- `Durbin-Watson`: Статистика Дарбина-Уотсона для проверки автокорреляции остатков.

- `Jarque-Bera (JB)`, `Prob(JB)`: Тест Жарка-Бера на нормальность распределения остатков.

- `Skew`, `Kurtosis`: Коэффициенты асимметрии и эксцесса остатков.

- `Cond. No.`: Число обусловленности матрицы независимых переменных.

Интерпретация результатов:

- Все коэффициенты (`X1`, `X2`, `X3`) статистически значимы, так как их p-значения меньше 0.05.

- Модель в целом адекватна, о чем свидетельствует высокое значение F-статистики и низкое p-значение и равное единице значение коэффициента детерминации `R-squared`

- Диагностические статистики указывают на отсутствие проблем с нормальностью и автокорреляцией остатков.

Таким образом, результаты регрессионного анализа показывают, что модель является статистически значимой и хорошо описывает данные. Коэффициенты при независимых переменных также являются значимыми, что позволяет сделать вывод об их влиянии на зависимую переменную.

3.3.4. Планирование эксперимента при регрессионном анализе

Как видно из общей формулы вычисления коэффициентов уравнения регрессии (158), процесс вычисления коэффициентов связан с необходимостью вычисления обратной матрицы $\left(X^T X\right)^{-1}$. Однако, как известно из курса линейной алгебры, обратные матрицы существуют не для всех матриц. Вырожденная матрица, определитель которой равен нулю, не имеет обратной. Кроме того, существуют плохо обусловленные матрицы, определитель которых не равен, но близок к нулю. Поскольку в формулу обратной матрицы определитель входит в знаменателе, то если матрица $\left(X^T X\right)$ окажется плохо обусловленной, коэффициенты уравнения регрессии b будут вычислены с большой ошибкой.

Отсюда следует, что точность оценок коэффициентов уравнения регрессии, а также сама возможность их вычисления зависит в конечном итоге от свойств матрицы *X* (154), элементами которой являются значения независимых переменных во всех опытах.

В связи с этим обстоятельством различают два способа постановки эксперимента. Первый способ - так называемый *пассивный* эксперимент, при котором исследователь или вообще лишен возможности независимо изменять значения входных переменных, а только фиксирует их значения в каждом опыте или же устанавливает их значения, не учитывая при этом свойств получающейся матрицы *X*. Пассивный эксперимент обладает недостатком, связанным с опасностью получить плохо обусловленную или вырожденную матрицу $\left(X^T X\right)$. Вместе с тем, иногда он является единственно возможным. Например, такая ситуация возникает, когда нужно получить уравнение регрессии, описывающее связь между физико-химическими характеристиками сырья и каким-либо свойством композиции. Сырье поступает на предприятие в готовом виде, и менять его характеристики не представляется возможным - можно лишь их контролировать для каждой партии. В тех же случаях, когда исследователь может независимо устанавливать значения входных переменных в каждом опыте, целесообразно использовать другой способ постановки эксперимента - *активный* или *планируемый* эксперимент [30, 31, 32] .

Существует два различных способа планирования эксперимента - *априорное планирование* и *апостериорное* или *последовательное* планирование.

В первом случае исследователь как бы заранее ограничивает область изменения независимых переменных и изменяет их значения внутри этой области. На основании результатов эксперимента в данной области рассчитывается уравнение регрессии, описывающее всю область исследования целиком. Дальнейшие выводы делаются на основе анализа данного уравнения. Этот способ применяется в тех случаях, когда уже перед началом исследования имеется определенная (априорная) информация об области, в которой целесообразно изменять значения независимых факторов. Кроме того, данный способ используется, когда нужно получить математическую модель, описывающую явление в широком диапазоне изменения входных переменных.

Однако иногда исследователя интересует не вся область допустимых значений факторов, а важно лишь найти оптимальную точку, в которой сочетание факторов обеспечивает наиболее желательное значение отклика. В этих случаях целесообразно не тратить лишних усилий на постановку эксперимента во всей области допустимых значений входных переменных, а поставить только минимальное количество опытов, необходимое для нахождения оптимальной точки. Такую возможность предоставляет метод последовательного планирования. В нем опыты делаются последовательно. На каждом шаге анализируется информация, полученная в результате опыта, и делается вывод о направлении дальнейшего изменения входных переменных. Данный метод особенно эффективен, когда каждый опыт занимает длительное время и цена его высока.

Часто целесообразно сочетать оба метода. Сначала с помощью последовательного планирования найти область наиболее желательного сочетания факторов. Ее называют также оптимальной областью или областью, близкой к экстремуму. Затем методом априорного планирования поставить в данной области эксперимент с целью описания ее уравнением регрессии второго порядка типа (147) и с помощью анализа данного уравнения решить задачу оптимизации.

Поскольку, с одной стороны, некоторые из методов последовательного планирования сами включают элементы априорного планирования (метод Бокса-Уилсона), а с другой стороны, последовательное планирование логически примыкает к методам решения задачи оптимизации, ниже будут рассмотрены только методы априорного планирования эксперимента, а методы последовательного планирования (метод Бокса- Уилсона и симплексный метод) будут рассмотрены в разделе, посвященном решению задачи оптимизации.

3.3.5. Планы первого порядка

Для линейных регрессионных моделей (149, 150) используют планы первого порядка или линейные планы. План эксперимента фактически представляет собой матрицу независимых переменных (154), уровни которых выбираются таким образом, чтобы обеспечить требуемые свойства матрицы. Из линейных планов шире всего используются ортогональные планы, в которых уровни независимых переменных x_{ij} выбираются такими, чтобы обеспечить ортогональность матрицы *X*.

Для удобства расчетов уровни независимых переменных переводят из натуральных единиц в условные. Чтобы осуществить такой переход, необходимо для каждой из независимых переменных назначить некоторое среднее значение $\left(x_j^0\right)$, относительно которого она изменяется (варьируется), и шаг варьирования $\left(\Delta x_j\right)$. Перевод осуществляется по формуле

$$x_j^{усл} = \frac{x_j^{нат} - x_j^0}{\Delta x_j} \quad . \qquad (166)$$

В линейных планах независимые переменные обычно варьируют на двух уровнях. Это связано с тем, что для вычисления коэффициентов линейного уравнения регрессии достаточно двух точек на каждую независимую переменную. При выбранных значениях x_j^0 и Δx_j максимальное значение независимых переменных в условных единицах будет равно +1, а минимальное будет равно -1. Простейшим планом эксперимента является так называемый полный факторный эксперимент. В нем независимые переменные принимают все возможные сочетания значений. Число опытов в таком плане равно $N = 2^k$, где k - число независимых факторов. Для двух факторов матрица планирования полного факторного эксперимента включает четыре опыта и имеет вид, показанный в табл. 5.

Таблица 5. Матрица линейного плана для двух переменных

№ опыта	x_1	x_2	y
1	-1	-1	y_1
2	+1	-1	y_2
3	-1	+1	y_3
4	+1	+1	y_4

Для трех независимых переменных число опытов будет равно восьми, и матрица планирования будет иметь вид, показанный в табл. 6.

Таблица 6. Матрица линейного плана для трех переменных

№ опыта	x_1	x_2	x_3	y
1	-1	-1	-1	y_1
2	+1	-1	-1	y_2
3	-1	+1	-1	y_3
4	+1	+1	-1	y_4
5	-1	-1	+1	y_5
6	+1	-1	+1	y_6
7	-1	+1	+1	y_7
8	+1	+1	+1	y_8

Коэффициенты уравнения регрессии в активном эксперименте могут быть вычислены так же, как и пассивном, по общей формуле (158), однако в случае линейных планов матрица планирования полного факторного эксперимента обладает свойством ортогональности

$$\begin{matrix} \sum_{i=1}^{N} x_{ij}^2 = N \\ \sum_{i=1, j\neq l}^{N} x_{ij} x_{il} = 0 \end{matrix} \quad . \qquad (167)$$

Для такой матрицы формулы для вычисления коэффициентов существенно упрощаются и принимают вид

$$b_j = \frac{\sum_{i=1}^{N} x_{ij} y_i}{\sum_{i=1}^{N} x_{ij}^2} \quad . \qquad (168)$$

Поскольку значения x_{ij} в каждом опыте равны +1 или -1, то по существу для вычисления коэффициента b_j по формуле (168) надо просто просуммировать значения откликов y_i со знаками, соответствующими столбцу x_j, и разделить полученную сумму на N. Для вычисления коэффициента b_0 в матрицы планирования (табл. 5, 6) надо добавить по одному столбцу x_0, все значения элементов которого положить тождественно равными +1. Тогда b_0 можно будет вычислить, как и остальные коэффициенты, по формуле (168).

Ортогональный план гарантирует линейную независимость столбцов матрицы X, что обеспечивает вычисление оценок коэффициентов регрессии с максимальной точностью. Следует заметить, что если в качестве элементов матрицы X выбирать независимые равномерно распределенные случайные числа, то столбцы матрицы также будут линейно независимы, чем будет обеспечена ее невырожденность. В качестве иллюстрации ниже приведена программа, где

коэффициенты уравнения регрессии вычисляются по ортогональной матрице X линейного плана, представленного в таблице 6 и по матрице X1, элементы которой случайные числа, равномерно распределенные в интервале [-1,1].

```
import numpy as np
from numpy.linalg import inv
import scipy.stats as stats
#Матрица планированного эксперимента
X=np.array(
[[-1,-1,-1],
 [ 1,-1,-1],
 [-1, 1,-1],
 [ 1, 1,-1],
 [-1,-1, 1],
 [ 1,-1, 1],
 [-1, 1, 1],
 [ 1, 1, 1]])
N=np.shape(X)[0] #Объем выборки
k=np.shape(X)[1] #Количество коэффициентов, не считая b_0
sigma=0.5 #Стандартное отклонение ошибки отклика
alpha=0.05 #Уровень значимости
#Генерируем случайную матрицу независимых переменных
X1=np.random.rand(N,k)*2-1
#Добавляем в матрицы столбец для свободного члена
X=np.hstack([np.ones((N,1)),X])
X1=np.hstack([np.ones((N,1)),X1])
dof=N-(k+1)
#Функция для генерирования ошибки опыта
def get_yerr(y,sigma,N):
    err=np.random.normal(0, sigma, N)
    S2err=np.mean(err**2)
    return(y+err,S2err)
```

```
#Функция для расчета коэффициентов регрессии и F-статистики
def calc_reg(y,X):
    y_err,S2err=get_yerr(y,sigma,N)
    hat_b=inv(X.T @ X)@(X.T @ y_err)
    y_calc=X @ hat_b
    S2=((y_calc-y_err) @ (y_calc-y_err))/dof
    F_stat=S2/(S2err)
    return(hat_b,F_stat)
#Задаем истинные значения коэффициентов единицами
b=np.ones(k+1)
#Вычисляем истинные значения откликов для матриц X и X1
y=X @ b
y1=X1 @ b
#Вычисляем коэффициенты регрессии и F-статистики для матриц X и X1
hat_b_X,F_stat_X=calc_reg(y,X)
hat_b_X1,F_stat_X1=calc_reg(y1,X1)
print("Коэффициенты регрессии и F-статистика для матрицы X:")
print(hat_b_X,F_stat_X)
print("Коэффициенты регрессии и F-статистика для матрицы X1:")
print(hat_b_X1,F_stat_X1)
f_critical = stats.f.ppf(1 - alpha, N-k, N-1)
print(f"Критическое значение F: {f_critical}")
```

В этой программе коэффициенты регрессии для наглядности заданы единицами и для вычисления оценок коэффициентов методом наименьших квадратов создана функция calc_reg(y,X), возвращающая оценки коэффициентов и F-статистику.

Результат выполнения программы:

```
Коэффициенты регрессии и F-статистика для матрицы X:
[0.90855154 0.84357914 1.02379347 1.36662125] 0.33663964964047166
Коэффициенты регрессии и F-статистика для матрицы X1:
```

[0.93648285 0.98047742 1.79625802 0.95838035] 0.652218424386365

Критическое значение F: 3.9715231506113415

Как видно, и матрица ортогонального плана X и случайная матрица X1 обеспечивают адекватные регрессионные модели.

Полный факторный эксперимент позволяет вычислить коэффициенты не только линейного уравнения регрессии, но также и обобщенного линейного уравнения, включающего эффекты взаимодействия факторов:

$$y = b_0 + \sum_j b_j x_j + \sum_{i<j} b_{ij} x_i x_j \quad . \qquad (169)$$

Для вычисления эффектов взаимодействия b_{ij} в матрицу планирования необходимо добавить столбцы $x_i x_j$, отвечающие данным коэффициентам. Элементы данных столбцов получаются произведением элементов столбцов x_i и x_j. Для трех факторов матрица планирования обобщенного линейного уравнения, включающая столбец для вычисления свободных членов, имеет вид, представленный в табл. 7. В силу ортогональности матрицы планирования коэффициенты, отвечающие эффектам взаимодействия факторов b_{ij}, могут быть вычислены по формуле (168).

Таблица 7. Матрица планирования обобщенного линейного плана для трех факторов

№ опыта	x_0	x_1	x_2	x_3	$x_1 x_2$	$x_1 x_3$	$x_2 x_3$	y
1	+1	-1	-1	-1	+1	+1	+1	y_1
2	+1	+1	-1	-1	-1	-1	+1	y_2
3	+1	-1	+1	-1	-1	+1	-1	y_3
4	+1	+1	+1	-1	+1	-1	-1	y_4
5	+1	-1	-1	+1	+1	-1	-1	y_5
6	+1	+1	-1	+1	-1	+1	-1	y_6
7	+1	-1	+1	+1	-1	-1	+1	y_7
8	+1	+1	+1	+1	+1	+1	+1	y_8

Эффекты взаимодействия показывают, как изменяется влияние одного из факторов при изменении другого. Помимо парных эффектов взаимодействия в многофакторных моделях могут возникать эффекты взаимодействия более высокого порядка, например тройные. В план эксперимента все эффекты взаимодействия вводятся аналогичным образом - путем добавления столбцов, элементы которых равны произведению столбцов факторов, отвечающих данному взаимодействию. Однако в некоторых случаях по физическому смыслу модели заранее известно, что некоторые из эффектов взаимодействия должны быть несущественными (незначимыми). В такой ситуации появляется возможность увеличить количество одновременно исследуемых факторов без увеличения числа опытов. Это достигается путем постановки плана по схеме так называемого дробного факторного эксперимента. В дробном факторном эксперименте новые факторы вводятся вместо незначимых эффектов взаимодействия так, что уровни нового фактора выбираются равными значениям столбца эффекта взаимодействия, вместо которого вводится новый фактор. Например, если для трехфакторного эксперимента взаимодействие между факторами x_1 и x_3 незначимо, то вместо эффекта $x_1 x_3$ в план эксперимента можно

ввести новый фактор x_4 , и при том же количестве опытов изучать одновременно влияние не трех независимых переменных, а четырех. Матрица планирования в этом случае примет вид, представленный в табл. 8.

Таблица 8. Матрица планирования дробного факторного эксперимента

№ опыта	x_0	x_1	x_2	x_3	$x_1 x_2$	x_4	$x_2 x_3$	y
1	+1	-1	-1	-1	+1	+1	+1	y_1
2	+1	+1	-1	-1	-1	-1	+1	y_2
3	+1	-1	+1	-1	-1	+1	-1	y_3
4	+1	+1	+1	-1	+1	-1	-1	y_4
5	+1	-1	-1	+1	+1	-1	-1	y_5
6	+1	+1	-1	+1	-1	+1	-1	y_6
7	+1	-1	+1	+1	-1	-1	+1	y_7
8	+1	+1	+1	+1	+1	+1	+1	y_8

Все коэффициенты уравнения регрессии вычисляются по формуле (168). Максимально в дробном факторном эксперименте можно ввести столько дополнительных факторов, сколько имеется эффектов взаимодействия. Такой план называется насыщенным. Необходимо помнить при этом, что все коэффициенты факторов, вводимых в план вместо эффектов взаимодействия, оказываются смешанными с соответствующими эффектами взаимодействия, и относиться к интерпретации коэффициентов с осторожностью.

3.3.6. Планы второго порядка

Для вычисления коэффициентов уравнения параболической регрессии вида

$$y = b_0 + \sum_{i=1}^{k} b_i x_i + \sum_{i=1}^{k} b_{ii} x_i^2 + \sum_{i,j;i<j}^{k} b_{ij} x_i x_j \qquad (170)$$

применяют планирование второго порядка.

Планы второго порядка отличаются от линейных тем, что факторы необходимо варьировать более чем на двух уровнях, минимум на трех. Простейшим планом второго порядка является полный факторный эксперимент, при котором реализуются все возможные сочетания факторов на всех уровнях. Для трех уровней число опытов такого плана составляет $N = 3^k$. С ростом числа факторов *k* число опытов быстро возрастает. Поэтому были разработаны более экономичные, так называемые композиционные планы.

Композиционный план состоит из нескольких групп опытов (нескольких блоков), включающих:

- точки полного или дробного факторного эксперимента линейного плана, отвечающего данному числу факторов с числом опытов $N_ф = 2^k$;
- "звездные точки" (план типа "креста"), при котором все переменные, кроме одной, закрепляются на среднем уровне, а оставшаяся варьируется с шагом α . Число опытов в данном блоке составляет $N_\alpha = 2k$;
- опыты в центре плана (нулевые или центральные точки); число опытов в данной группе может изменяться в зависимости от вида плана и составляет в общем случае N_0 .

Общее число опытов композиционного плана составляет $N = N_ф + N_\alpha + N_0$. Наибольшее распространение получили два вида композиционных планов - ортогональные центральные композиционные планы (ОЦКП) и рототабельные центральные композиционные планы (РЦКП).

При ортогональном планировании матрица планирования эксперимента обладает свойством ортогональности. В силу этого оценки коэффициентов уравнения регрессии вычисляются независимо друг от друга с минимальными дисперсиями. Незначимые коэффициенты можно сразу

отбрасывать из уравнения регрессии без пересчета остальных значимых коэффициентов, как это необходимо при неортогональных планах.

Рототабельные планы позволяют получить уравнение регрессии, предсказывающее значение отклика с одинаковой точностью во всех направлениях на одинаковом расстоянии от центра плана.

Ниже (табл. 9, 10) приводятся характеристики блоков опытов ортогональных и рототабельных планов.

Таблица 9. Параметры ортогональных планов второго порядка

Число факторов, *k*	Величина звездного плеча, α	N_α	N_0	N_ϕ	N
2	1,0	4	1	4	9
3	1,215	6	1	8	15
4	1,414	8	1	16	25

Таблица 10. Параметры рототабельных планов второго порядка

Число факторов, *k*	Величина звездного плеча, α	N_α	N_0	N_ϕ	N
2	1,414	4	5	4	13
3	1,682	6	6	8	20
4	2,000	8	7	16	31

Как видно, различные типы планов отличаются только величиной звездного плеча α и числом опытов в центре плана. Ортогональные планы несколько экономичнее с точки зрения числа опытов. Кроме того, в ортогональных планах коэффициенты уравнения регрессии вычисляются по более простым формулам (168). Это преимущество, однако, на сегодняшний день практически не существенно, поскольку использование современных программных пакетов позволяет полностью автоматизировать расчет коэффициентов с применением общей формулы регрессионного анализа (143). Сегодня нет необходимости также и отбрасывать из уравнения незначимые коэффициенты, поскольку раньше это делалось с целью сокращения трудоемкости расчетов функции отклика, а применение ЭВМ снимает эту проблему. Поэтому в настоящее время при выборе типа плана наибольшее значение приобретают свойства плана, связанные с точностью предсказания функции отклика, а не с простотой ее вычисления. В этом отношении рототабельные планы являются более предпочтительными, так как позволяют получить уравнение регрессии, наилучшим образом описывающее исследуемую область.

Пользуясь параметрами планов (табл. 9, 10), можно легко построить матрицу планирования для выбранного числа факторов. Для примера ниже (табл. 11, 12) приводятся матрицы планирования ортогонального и рототабельного планов для двух факторов.

Таблица 11. Матрица планирования ОЦКП для двух факторов

Вид блока плана	№ опыта	x_1	x_2	y
$N_ф$	1	-1	-1	y_1
	2	+1	-1	y_2
	3	-1	+1	y_3
	4	+1	+1	y_4
$N_α$	5	-1	0	y_5
	6	+1	0	y_6
	7	0	-1	y_7
	8	0	+1	y_8
N_0	9	0	0	y_9

Таблица 12. Матрица планирования РЦКП для двух факторов

Вид блока плана	№ опыта	x_1	x_2	y
$N_ф$	1	-1	-1	y_1
	2	+1	-1	y_2
	3	-1	+1	y_3
	4	+1	+1	y_4
$N_α$	5	-1,414	0	y_5
	6	+1,414	0	y_6
	7	0	-1,414	y_7
	8	0	+1,414	y_8
N_0	9	0	0	y_9
	10	0	0	y_{10}
	11	0	0	y_{11}
	12	0	0	y_{12}
	13	0	0	y_{13}

Статистический анализ уравнений регрессии в планированном эксперименте проводят так же, как и обычно в регрессионном анализе. Ниже приводится пример программы в котором приводится пример расчета коэффициентов уравнения двухфакторной параболической регрессии с использованием матриц ортогонального (табл. 11) и рототабельного планов (табл. 12).

```
import numpy as np
from numpy.linalg import inv
```

```
import scipy.stats as stats
np.random.seed(0)
#Матрица ортогонального плана
Xort=np.array(
[[-1,-1],
 [ 1,-1],
 [-1, 1],
 [ 1, 1],
 [-1, 0],
 [1,  0],
 [0, -1],
 [0,  1],
 [0,  0]])
#Матрица рототабельного плана
Xrot=np.array(
[[-1,       -1],
[ 1,       -1],
[-1,         1],
[ 1,         1],
[-1.414,         0],
[ 1.414,         0],
[ 0,   -1.414],
[ 0,      1.414],
[ 0,         0],
[ 0,         0],
[ 0,         0],
[ 0,         0],
[ 0,         0]])
def ad_cols(Xexp):
    N=np.shape(Xexp)[0]
    X_lin=Xexp[:,:2]
```

```
    X0=np.ones((N,1))
    X=np.hstack((X0,X_lin))
    X=np.column_stack((X,X[:,1]**2,X[:,2]**2,X[:,1]*X[:,2]))
    return(X)
Xort=ad_cols(Xort)
Xrot=ad_cols(Xrot)
Nort=np.shape(Xort)[0] #Объем выборки Nort
Nrot=np.shape(Xrot)[0] #Объем выборки Nrot
k=np.shape(Xort)[1] #Количество коэффициентов
sigma=0.5 #Стандартное отклонение ошибки отклика
alpha=0.05 #Уровень значимости
#Функция для генерирования ошибки опыта
def get_yerr(y,sigma,N):
    err=np.random.normal(0, sigma, N)
    S2err=np.mean(err**2)
    return(y+err,S2err)
#Функция для расчета коэффициентов регрессии и F-статистики
def calc_reg(y,X):
    N=np.shape(X)[0]
    dof=N-k
    y_err,S2err=get_yerr(y,sigma,N)
    hat_b=inv(X.T @ X)@(X.T @ y_err)
    y_calc=X @ hat_b
    S2=((y_calc-y_err) @ (y_calc-y_err))/dof
    F_stat=S2/(S2err)
    return(hat_b,F_stat)
#Задаем истинные значения коэффициентов
b=np.ones(k)
#Вычисляем истинные значения откликов для матриц Xort и Xrot
yort=Xort @ b
yrot=Xrot @ b
```

```
#Вычисляем коэффициенты регрессии и F-статистики для матриц Xort и Xrot
hat_b_Xort,F_stat_Xort=calc_reg(yort,Xort)
hat_b_Xrot,F_stat_Xrot=calc_reg(yrot,Xrot)
print("Коэффициенты регрессии и F-статистика для матрицы Xort:")
print(hat_b_Xort,f"{F_stat_Xort:.4f}")
print("Коэффициенты регрессии и F-статистика для матрицы Xrot:")
print(hat_b_Xrot,f"{F_stat_Xrot:.4f}")
f_crit_ort = stats.f.ppf(1 - alpha, Nort-k, Nort-1)
print(f"Критическое значение Fort: {f_crit_ort:.4f}")
f_crit_rot = stats.f.ppf(1 - alpha, Nrot-k, Nrot-1)
print(f"Критическое значение Fort: {f_crit_rot:.4f}")
```

Пример вывода результатов:

```
Коэффициенты регрессии и F-статистика для матрицы Xort:
[0.85989376 0.75445202 0.99616467 1.40692468 1.38403745 1.32825629] 0.7294
Коэффициенты регрессии и F-статистика для матрицы Xrot:
[0.73543858 0.9684817  1.20637375 1.21476595 1.37259051 0.94666489] 0.9364
Критическое значение Fort: 4.0662
Критическое значение Fort: 2.9134
```

Как видно, обе модели адекватны экспериментальным данным.

3.3.7. Уравнение регрессии на ортогональных функциях

В предыдущем разделе показаны преимущества планов эксперимента в виде ортогональных матриц, основным из которых является линейная независимость столбцов матрицы планирования, что обеспечивает ее невырожденность и удобство вычисления оценок коэффициентов регрессии. Вместе с тем и другая возможность обеспечения линейной независимости оценок коэффициентов регрессии, заключающаяся в использовании в качестве уравнения регрессии ортогональных функций.

Ортогональные функции играют важную роль в математике, особенно в теории рядов Фурье и линейных операторов. Две функции $f(x)$ и $g(x)$ называются ортогональными на интервале $[a,b]$, если их скалярное произведение равно нулю. Это условие можно выразить через интеграл

$$\int_a^b f(x)\,g(x)\,dx = 0 \qquad (171)$$

Если функции являются вещественными, то это условие определяет их ортогональность на заданном интервале. Ортогональные функции могут быть как действительными, так и комплексными, и они часто используются для разложения функций в ряды, что позволяет упростить анализ различных математических задач.

Среди различных ортогональных функций наиболее известными являются тригонометрические: $\sin(x)$ и $\cos(x)$, которые являются ортогональными на интервале $[0,2\pi]$

$$\int_0^{2\pi} \sin(nx)\sin(mx)\,dx = 0 \quad , \quad n \neq m \qquad (172)$$

$$\int_0^{2\pi} \cos(nx)\cos(mx)\,dx = 0 \quad , \quad n \neq m \qquad (173)$$

Другим примером ортогональных функций являются ортогональные полиномы. Ортогональные полиномы представляют собой последовательность многочленов, которые обладают свойством ортогональности относительно некоторого скалярного произведения. Это означает, что для двух различных полиномов $P_n(x)$ и $P_m(x)$ (где $n \neq m$) выполняется условие

$$\int_a^b P_n(x)\,P_m(x)\,w(x)\,dx = 0 \quad , \qquad (174)$$

где $w(x)$ — весовая функция, а $[a,b]$ — интервал ортогональности.

Среди наиболее известных систем ортогональных полиномов можно выделить: полиномы Чебышёва, полиномы Лагерра, полиномы Эрмита, полиномы Якоби и полиномы Лежандра.

Многочлены Лежандра $L_n(x)$ определяются рекурсивно или через формулу Родрига

$$L_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left((x^2-1)^n\right) \quad . \qquad (175)$$

Они удовлетворяют условию ортогональности (174) на интервале $[-1,1]$ с весовой функцией w(x)=1

$$\int_{-1}^{1} L_n(x)\,L_m(x)\,dx=0 \quad . \tag{176}$$

Ниже приведены первые десять полиномов Лежандра

$$\begin{gathered} L_0=1 \\ L_1=x \\ L_2=\left(3x^2-1\right)/2 \\ L_3=\left(5x^3-3x\right)/2 \\ L_4=\left(35x^4-30x^2+3\right)/8 \\ L_5=\left(63x^5-70x^3+15x\right)/8 \\ L_6=231/16\,x^6-315/16\,x^4+105/16\,x^2-5/16 \\ L_7=429/16\,x^7-693/16\,x^5+315/16\,x^3-35/16\,x \\ L_8=6435/128\,x^8-3003/32\,x^6+3465/64\,x^4-315/32\,x^2+35/128 \\ L_9=12155/128\,x^9-6435/32\,x^7+9009/64\,x^5-1155/32\,x^3+315/128\,x \end{gathered} \tag{177}$$

Применение ортогональных функций в численных методах связано с разложением функций в ряды по ортогональной системе функций. Выше отмечалось, что использование уравнения регрессии в форме многочлена (146) обосновывается возможностью разложения непрерывной функции $f(x)$ в степенной ряд Тейлора вблизи определенной точки a

$$f(x)=\sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n \quad , \tag{178}$$

где где $f^{(n)}(a)$ — $n-я$ производная функции в точке a .

В отличие от ряда Тейлора, разложение функции в ряд по ортогональным многочленам использует систему ортогональных функций для представления функции. Разложение имеет вид

$$f(x)=\sum_{n=0}^{\infty}c_n\,\phi_n(x) \quad , \tag{179}$$

где $\phi_n(x)$ - ортогональные функции (например, ортогональные многочлены), а коэффициенты c_n вычисляются по формулам

$$c_n = \frac{\int_a^b f(x)\,\phi_n(x)\,dx}{\int_a^b \phi_n(x)\,\phi_n(x)\,dx} \quad . \qquad (180)$$

Таким образом, если коэффициенты уравнения регрессии в виде обычного многочлена (146) можно рассматривать как производные некоторой функции регрессии $f(x)$ в точке $a=0$, то коэффициенты уравнения регрессии в виде ортогонального многочлена представляют собой согласно (180) средние значения данной функции на отрезке $[a,b]$ по распределениям $\phi_n(x)$. Последний вариант, как будет показано в примерах программ ниже, имеет определенные преимущества. Ниже приведен пример программы, иллюстрирующей разложение функции $f(x)=\sin(x\pi)$ в степенной ряд Тейлора до члена x^5 включительно и в ряд по ортогональным многочленам Лежандра до $n=5$ (6 членов: $n=0,1,2,3,4,5$) на отрезке $[-1,1]$.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import legendre
# Определяем функцию
def f(x):
    return np.sin(x * np.pi)
# Разложение в ряд Тейлора до x^5
def taylor_series(x):
    return (np.pi * x - (np.pi**3 / 6) * x**3 + (np.pi**5 / 120) * x**5)
# Ортогональные многочлены Лежандра
def legendre_series(x, n_terms):
    result = 0
    for n in range(n_terms):
        L_n = legendre(n)  # Получаем n-й многочлен Лежандра
        # Вычисляем коэффициент a_n
        a_n = (2 * n + 1) / 2 * np.trapz(f(np.linspace(-1, 1, 1000)) * L_n(np.linspace(-1, 1, 1000)),
                                         np.linspace(-1, 1, 1000))
        result += a_n * L_n(x)
    return result
```

```
# Создаем массив значений x
x_values = np.linspace(-1, 1, 400)
f_values = f(x_values)
# Вычисляем приближения
taylor_values = taylor_series(x_values)
legendre_values = legendre_series(x_values, n_terms=6)  # 6 членов (n=0,1,2,3,4,5)
# Построение графиков
plt.figure(figsize=(12, 6))
# График функции
plt.plot(x_values, f_values, label='sin(x * π)', color='green', linewidth=2)
# График разложения в ряд Тейлора
plt.plot(x_values, taylor_values, label='Ряд Тейлора (до x^5)', color='blue', linestyle='--')
# График разложения по многочленам Лежандра
plt.plot(x_values, legendre_values, label='Ряд Лежандра (до n=5)', color='red', linestyle=':')
# Настройка графика
plt.title('Разложение функции sin(x * π) в ряд Тейлора и по многочленам Лежандра')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.legend()
plt.grid()
plt.xlim(-1, 1)
plt.ylim(-1.5, 1.5)
# Показать график
plt.show()
```

Вычисление коэффициентов многочлена Лежандра осуществляется с помощью функции legendre(n), библиотеки scipy.special. Для интегрирования в формуле (180) используется численный метод трапеций с использованием функции trapz библиотеки numpy. Графики функции и ее приближений приведены на рисунке 14.

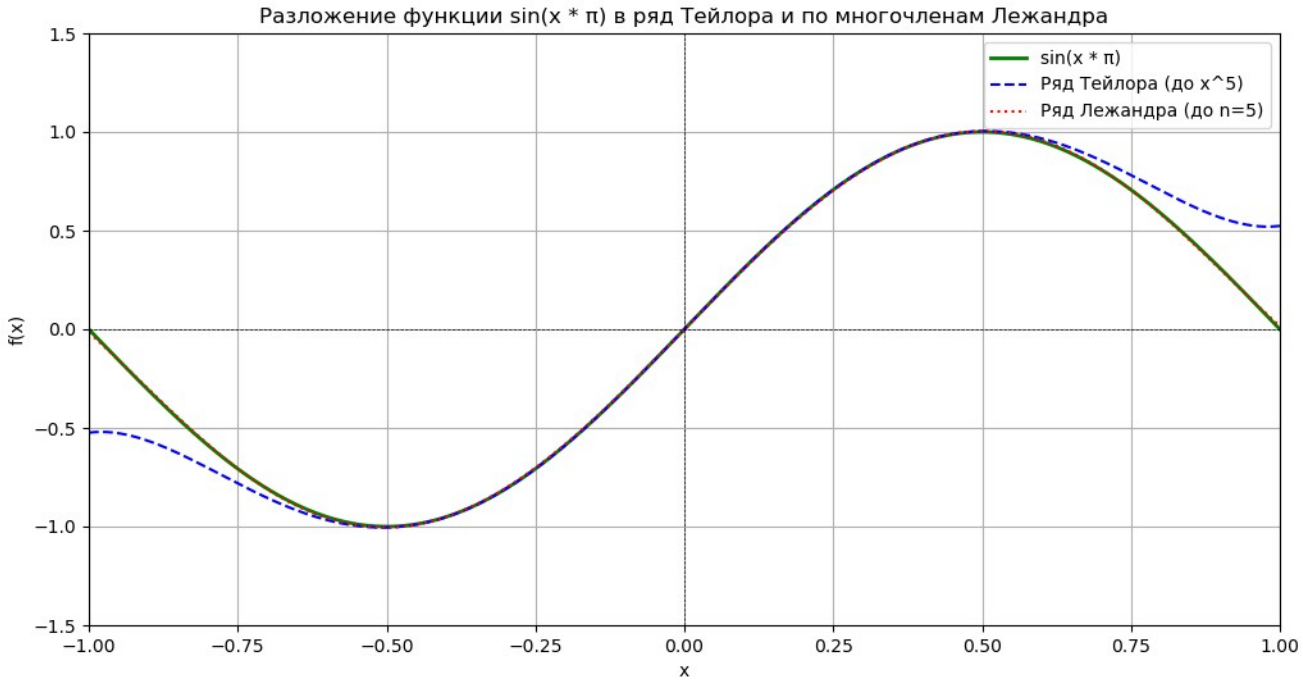

Рисунок 14. Сравнение приближений функции $f(x)=\sin(x\pi)$ с помощью ряда Тейлора и ортогональных многочленов Лежандра. График ряда Лежандра практически совпадает с графиком функции $f(x)$.

Как можно видеть, графики функции $f(x)$ и ее приближения многочленом Лежандра почти совпадают, в то время как для разложения Тейлора с той же степенью полинома график приближения расходится с графиком функции на краях отрезка. Вместе с тем, если использовать обычный полином и полином Лежандра в качестве функций регрессии, то в обоих случаях для вычисления оценок коэффициентов используется метод наименьших квадратов, то есть коэффициенты вычисляются в результате усреднения невязок по всему интервалу значений функции. Поэтому разницы в точности аппроксимации между этими методами уже не будет, но сохранится разница в оценках коэффициентов. Это иллюстрирует приведенная ниже программа, в которой та же самая функция $f(x)=\sin(x\pi)$ моделируемая со случайной погрешностью, аппроксимируется рассмотренными выше полиномами с разной степенью вплоть до x^5 включительно.

```
import numpy as np
from numpy.linalg import inv
import scipy.stats as stats
```

```
import matplotlib.pylab as plt
np.random.seed(0)
sigma=0.01 #Стандартное отклонение ошибки отклика
alpha=0.05 #Уровень значимости
N=20 # Объем выборки
k=5 #Степень полинома
dof=N-(k+1)
def get_yerr(y,sigma,N):
    err=np.random.normal(0, sigma, N)
    S2err=np.mean(err**2)
    return(y+err,S2err)
#Модель системы
def model1(x):
    y=np.sin(x*np.pi)
    return (y)
x=np.linspace(-1,1,N)
np.reshape(x,(N,1))
y=model1(x)
yerr,S2err=get_yerr(y,sigma,N)
#Расчет матрицы для простого полинома
def make_poly_X(N,k):
    X0=np.ones((N,1))
    x=np.reshape(np.linspace(-1,1,N),(N,1))
    X=np.hstack((X0,x))
    for i in range(2,k+1):
        X=np.column_stack((X,X[:,1]**i))
    return(X)
#Расчет коэффициентов регрессии для простого полинома
def calc_reg(ye,k):
    X=make_poly_X(N,k)
    hat_b=inv(X.T @ X)@(X.T @ ye)
```

```
    y_calc=X @ hat_b
    S2=((y_calc-ye) @ (y_calc-ye))/dof
    F_stat=S2/(S2err)
    return(hat_b,y_calc,F_stat)
#Функция, возвращающая полином Лежандра степени k<=9
def Lezh(x,k):
    y=np.zeros((10,len(x)))
    y[0,:]=1
    y[1,:]=x
    y[2,:]=(3*x** 2-1)/2
    y[3,:]=(5*x**3-3*x)/2
    y[4,:]=(35*x**4-30*x**2+3)/8
    y[5,:]=(63*x**5-70*x**3+15*x)/8
    y[6,:]=231/16*x**6-315/16*x**4+105/16*x**2-5/16
    y[7,:]=429/16*x**7-693/16*x**5+315/16*x**3-35/16*x
    y[8,:]=6435/128*x**8-3003/32*x**6+3465/64*x**4-315/32*x**2+35/128
    y[9,:]=12155/128*x**9-6435/32*x**7+9009/64*x**5-1155/32*x**3+315/128*x
    return(y[:k+1,:])
#Функция вычисляющая коэффициенты аппроксимационного многочлена Лежандра методом наименьших квадратов
def calc_L(x,y_err,k):
    X=Lezh(x.ravel(),k).T
    L=inv(X.T @ X)@(X.T @ y_err)
    y_calc=X @ L
    S2=((y_calc-y_err) @ (y_calc-y_err))/dof
    F_stat=S2/(S2err)
    return(L,y_calc,F_stat)
#Вывод результатов для простого и ортогонального полиномов
plt.plot(x,yerr,"o",label="функция")#График исходной модели
def format_float(number, decimal_places=4):
    return f"{number:.{decimal_places}f}"
```

```
print("Простой полином:")
for i in range(1,k+1):
    hat_b,y_calc,F_stat=calc_reg(yerr,i)
    formatted_hat_b = [format_float(num) for num in hat_b]
    print(formatted_hat_b,f"F_stat:{F_stat:.4f}")
colours=["red","green","blue","magenta","black","cyan"]#Цвета линий
print("Ортогональный полином:")
for i in range(1,k+1):
    L,y_calc,F_stat=calc_L(x,yerr,i)
    formatted_L = [format_float(num) for num in L]
    print(formatted_L,f"F_stat:{F_stat:.4f}")
    plt.plot(x,y_calc,colours[i-2],label=f"полином n={i}")
    plt.title('Приближение функции sin(x * π) уравнениями регрессии')
    plt.xlabel('x')
    plt.ylabel('Функция и регрессия')
    plt.legend()
plt.show()
f_crit = stats.f.ppf(1 - alpha, dof, N-1)
print(f"Критическое значение F: {f_crit:.4f}")
```

В данной программе значение функции моделируется со случайной ошибкой с тем, чтобы оценить адекватность различных приближений. Матрица X для простого полинома создается с помощью функции make_poly_X(N,k), а коэффициенты регрессии вычисляются с помощью функции calc_reg(ye,k). Для вычислений коэффициентов регрессии по полиномам Лежандра используются, соответственно функции Lezh(x,k) и calc_L(x,y_err,k). В конце программы производится сравнение коэффициентов и F-статистик для полиномов разных степеней и строятся графики значений функции и ее приближений полиномами разной степени. Результат работы программы:

```
Простой полином:
['0.0057', '0.8076'] F_stat:3192.0843
```

['0.0051', '0.8076', '0.0017'] F_stat:3192.0803

['0.0051', '2.6282', '0.0017', '-2.7546'] F_stat:84.4035

['0.0027', '2.6282', '0.0236', '-2.7546', '-0.0233'] F_stat:84.3432

['0.0027', '3.1035', '0.0236', '-4.7992', '-0.0233', '1.6903'] F_stat:1.4159

Ортогональный полином:

['0.0057', '0.8076'] F_stat:3192.0843

['0.0056', '0.8076', '0.0011'] F_stat:3192.0803

['0.0056', '0.9755', '0.0011', '-1.1018'] F_stat:84.4035

['0.0059', '0.9755', '0.0024', '-1.1018', '-0.0053'] F_stat:84.3432

['0.0059', '0.9484', '0.0024', '-1.1684', '-0.0053', '0.2146'] F_stat:1.4159

Критическое значение F: 2.2556

Как видно, точность аппроксимации простым полиномом и ортогональным полиномом одинакова. При этом адекватная аппроксимация достигается только при степени полинома x^5 . Разница между двумя способами аппроксимации проявляется лишь в значениях коэффициентов регрессии. Важно отметить, что в случае простого полинома с повышением максимальной степени полинома все коэффициенты регрессии изменяются. Это подчеркивает их скоррелированность (линейную зависимость). В то же время, в случае аппроксимации ортогональным полиномом оценки коэффициентов регрессии получаются практически не зависящими от степени полинома. При повышении степени полинома добавляется лишь новый коэффициент, а оценки предыдущих коэффициентов практически не меняются. На практике это обстоятельство оказывается весьма удобным. Графики аппроксимирующих функций для простого полинома и ортогонального полинома одинаковых степеней не различимы, поэтому на рисунке 15 приведены аппроксимирующие кривые только для полиномов Лежандра разных степеней. Видно, что адекватной аппроксимацией является лишь полином пятой степени, что соответствует статистике. Поскольку моделируемая функция нечетная, то коэффициенты при четной степени x близки к нулю, а графики полиномов четной степени совпадают с графиками нечетных степеней.

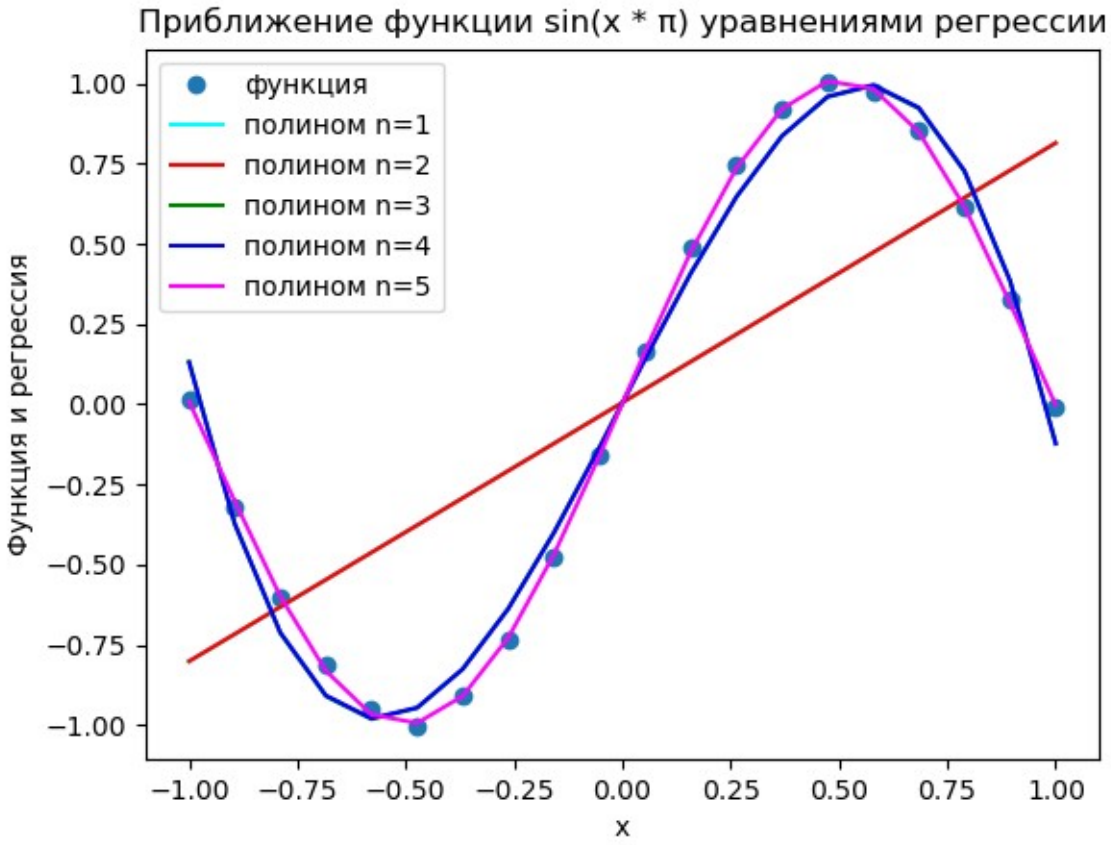

Рисунок 15. Графики смоделированных значений функции и аппроксимирующих полиномов разных степеней.

Глава 4. Методы решения задачи оптимизации

Главной задачей и конечной целью решения большинства инженерных проблем является обычно нахождение и поддержание экстремальных, т. е. наилучших показателей. Процесс нахождения и поддержания экстремальных значений целевой функции объекта и называется *оптимизацией*. Оптимизация технологических процессов является важной задачей для повышения эффективности производства. В литературе представлены различные методы оптимизации [33, 34, 35], которые можно разделить на аналитические и численные, а также на методы безусловной и условной оптимизации.

4.1. Постановка задачи оптимизации

Для того чтобы сформулировать задачу оптимизации, необходимо прежде всего выбрать целевую функцию, или критерий оптимизации y . В качестве такой функции может быть выбрана, например, производительность установки (для которой, очевидно, необходимо найти максимум), или энергозатраты процесса (для которых требуется найти минимум), или какой-либо из показателей качества получаемого продукта. Для математически корректной постановки задачи оптимизации критерий оптимизации должен быть один. На практике же обычно бывает одновременно несколько функций, описывающих процесс, к которым предъявляются различные, часто противоречивые, требования. В этой связи различают два варианта постановки задачи оптимизации.

Первый вариант - когда критерий оптимизации единственный, или же на основании нескольких функций можно сформировать такой единственный критерий - обобщенный критерий оптимизации (его называют также функцией желательности). Такой вариант постановки задачи оптимизации называется *безусловной оптимизацией*. В том случае, когда не удается выбрать одну из функций в качестве критерия оптимальности, можно попытаться скомбинировать из нескольких критериев один, например, отношение производительности установки к удельным энергозатратам. Другим способом является использование средневзвешенного критерия

$$y = \sqrt[k]{y_1^{\alpha 1} y_2^{\alpha 2} \dots y_k^{\alpha k}} \quad , \tag{181}$$

где y_i - частные критерии оптимальности, α_i - веса, подбираемые таким образом, что $\sum \alpha_i = k$. При этом, чем более важным является показатель y_i, тем более высоким назначается его вес α_i . Если какой-либо из показателей y_i, должен принимать минимальное значение, то его вес берется отрицательным.

Можно сформулировать также векторный критерий оптимальности $y(y_1, y_2 \dots y_k)$, для которого ищется экстремум в смысле той или иной нормы: $y = \|y\|$.

Второй вариант - *задача условной, или компромиссной оптимизации* - возникает тогда, когда необходимо найти экстремум целевой функции при наличии ограничений.

Если целевая функция или критерий оптимизации задана в виде

$$y = f(x) \ , \tag{182}$$

где $x = (x_1, x_2, \dots x_n)$ - вектор управляющих параметров, то оптимизация заключается в подборе такого значения $x^0 = (x_1^0, \dots, x_n^0)$, при котором y принимает оптимальное, т. е. минимальное или максимальное значение. Поиск такого значения целевой функции называют безусловной оптимизацией, подчеркивая при этом, что на независимые переменные не накладывается никаких ограничений (условий).

В отличие от безусловного оптимума, экстремум целевой функции, найденный при наличии ограничений на независимые переменные, называют условным или компромиссным экстремумом. Ограничения на независимые переменные могут иметь вид равенств или неравенств

$$\begin{gathered} \varphi_1(x) = 0 \ , \\ \varphi_2(x) \le 0 \ , \\ \dots\dots\dots\dots\dots \\ \varphi_p(x) \le 0 \ , \\ x_{j1} \le x_j \le x_{j2} \ , \quad j = 1..m \ . \end{gathered} \tag{183}$$

Такие ограничения могут возникать по конструктивным и технологическим требованиям. Задача компромиссной оптимизации в данном случае формулируется как отыскание значения $x^0=(x_1^0,\dots,x_n^0)$, обеспечивающего экстремум целевой функции (181) при наличии ограничений (183).

Таким образом, в общем случае постановка задачи оптимизации (поиск максимума) включает:

1) целевую функцию $f:U\to\mathbb{R},U\subset\mathbb{R}^n$;

2) множество допустимых значений независимого переменного, среди которых осуществляется поиск $X\subset U$.

Требуется найти вектор $x^0=(x_1^0,\dots,x_n^0)$, которому соответствует максимальное значение целевой функции на множестве X.

4.2. Аналитические методы определения экстремума

Аналитические методы определения безусловного экстремума основаны на теоремах математического анализа о необходимом и достаточном условиях экстремума функции многих переменных.

Функция $f:U\to\mathbb{R},U\subset\mathbb{R}^n$ имеет в точке $x^0=(x_1^0,\dots,x_n^0)$ локальный максимум (локальный минимум), если существует такая окрестность точки

x^0 , в которой при $x\neq x^0$ выполняется неравенство $f(x)<f(x^0)$, $(f(x)>f(x^0))$.

Теорема (необходимое условие экстремума). Если функция $f(x)$ имеет в точке $x^0=(x_1^0,\dots,x_n^0)$ локальный экстремум и в этой точке существует частная производная функции по аргументу x_i , то $\frac{\partial f}{\partial x_i}(x^0)=0$.

Следствие. Если функция $f(x)$ имеет в точке $x^0=(x_1^0,\dots,x_n^0)$ локальный экстремум и дифференцируема в этой точке, то

$$df(x^0)=\sum \frac{\partial f}{\partial x_i} dx_i(x^0)=0, \forall dx_i \quad . \qquad (184)$$

С использованием оператора градиента ∇ условие (184) может быть записано также в следующем виде:

$$\nabla y(x^0)=0 \quad . \qquad (185)$$

Прежде чем сформулировать теорему о достаточном условии экстремума, введем понятие квадратичной формы.

Функция вида $Q=\sum_{i,j=1}^{n} a_{ij} x_i x_j$, где a_{ij} - числа, причем $a_{ij}=a_{ji}$, называется квадратичной формой от переменных $x_1,\dots,x_n$.

Числа a_{ij} называются коэффициентами квадратичной формы, а составленная из этих коэффициентов симметричная матрица $A=(a_{ij})$ - матрицей квадратичной формы.

Квадратичная форма называется положительно определенной (отрицательно определенной), если для любых значений переменных $x_1,\dots,x_n$, одновременно не равных нулю, она принимает положительные (отрицательные) значения.

Квадратичная форма называется квазизнакоопределенной, если она принимает либо только неотрицательные, либо только неположительные значения.

Квадратичная форма называется знакопеременной, если она принимает как положительные, так и отрицательные значения.

Матрица A квадратичной формы называется положительно определенной и обозначается $A>0$, если $(Ax,x)>0$ для всех векторов $x\neq 0$. Аналогично вводятся также отрицательно, неотрицательно и неположительно определенные матрицы.

Между линейным пространством квадратичных форм и линейным пространством отвечающих им матриц имеется изоморфное соответствие. Следующие две теоремы определяют условия определенности матриц квадратичных форм.

1. Критерий Якоби. Для того чтобы матрица квадратичной формы была положительно определенной, необходимо и достаточно, чтобы все коэффициенты ее характеристического многочлена были отличны от нуля и имели чередующиеся знаки.

2. Для того чтобы матрица квадратичной формы удовлетворяла условию $A>0$ $(A\geqslant 0, A<0, A\leqslant 0)$, необходимо и достаточно, чтобы ее собственные значения λ_i удовлетворяли условию $\lambda_i>0 (\lambda_i\geqslant 0, \lambda_i<0, \lambda_i\leqslant 0)$.

Второй дифференциал функции в данной точке $x^0=(x_1^0,\dots,x_n^0)$

$$d^2 f(x^0)=\sum_{i,j=1}^{n}\frac{\partial^2 f}{\partial x_i \partial x_j}(x^0)\,dx_i\,dx_j \qquad (186)$$

является квадратичной формой от переменных $dx_1,\dots,dx_n$, а частные производные второго порядка $\frac{\partial f}{\partial x_i \partial x_j}(x^0)$ - коэффициентами этой квадратичной формы. Матрица H этой квадратичной формы называется матрицей Гессе.

Теорема. (Достаточное условие локального экстремума) Пусть функция $f(x)$ дифференцируема в некоторой окрестности точки $x^0=(x_1^0,\dots,x_n^0)$ и дважды дифференцируема в самой точке, причем $df(x^0)=0$. Тогда, если второй дифференциал $d^2 f(x^0)$ является положительно определенной (отрицательно определенной) квадратичной формой от переменных $dx_1,\dots,dx_n$, то функция $f(x)$ имеет в точке x^0 локальный минимум (максимум). Если же $d^2 f(x^0)$ является знакопеременной квадратичной формой, то в точке x^0 функция не имеет локального экстремума.

Таким образом, достаточным условием экстремума функции в критической точке является положительная или отрицательная определенность ее матрицы вторых производных (матрицы Гессе) в данной точке.

Причем, если матрица H является положительно определенной, то в точке экстремума функция достигает минимума, если отрицательно определенной - то максимума.

Пример 1. Найти точки локального экстремума функции $u(x_1, x_2, x_3) = 2x_1^2 - x_1 x_2 + 2x_1 x_3 - x_2 + x_2^2 + x_3^2$.

Ниже приведен текст программы с решением данной задачи.

```
import sympy as sp
# Определяем переменные
x, y, z = sp.symbols('x y z')
# Определяем функцию
u = 2*x**2 - x*y + 2*x*z - y + y**3 + z**2
# Вычисляем матрицу Гессе
H = sp.hessian(u, (x, y, z))
# Находим частные производные
du_dx = sp.diff(u, x)
du_dy = sp.diff(u, y)
du_dz = sp.diff(u, z)
# Находим критические точки, решая систему уравнений
solutions = sp.solve((du_dx, du_dy, du_dz), (x, y, z))
print(f"Решения системы уравнений (тип Rational): \n{solutions}")
critical_points=[]
# Преобразуем решения в тип float
for point in solutions:
    fpoint=[]
    for pt in point:
        fpoint.append(float(pt.evalf()))
    critical_points.append(fpoint)
print(f"Критические точки (тип float): \n{critical_points}")
# Анализируем матрицу Гессе в критических точках
for point in critical_points:
```

```
H_at_point = H.subs({x: point[0], y: point[1], z: point[2]})
# Находим собственные значения матрицы Гессе
eigenvalues = H_at_point.eigenvals()
print(f"\nМатрица Гессе в точке {point}:")
sp.pprint(H_at_point)
print(f"Собственные значения: {eigenvalues}")
# Определяем вид критической точки
if all(ev > 0 for ev in eigenvalues):
    print("Точка является локальным минимумом.")
elif all(ev < 0 for ev in eigenvalues):
    print("Точка является локальным максимумом.")
else:
    print("Точка является седловой точкой.")
```

В данной программе используется библиотека компьютерной алгебры sympy. В начале программы определяется символьная переменная u, которой присваивается символьное выражение заданной функции. Для вычисления матрицы Гессе

$$H=\begin{bmatrix} \frac{\partial^2 u}{\partial x^2} & \frac{\partial^2 u}{\partial x \partial y} & \frac{\partial^2 u}{\partial x \partial z} \\ \frac{\partial^2 u}{\partial y \partial x} & \frac{\partial^2 u}{\partial y^2} & \frac{\partial^2 u}{\partial y \partial z} \\ \frac{\partial^2 u}{\partial z \partial x} & \frac{\partial^2 u}{\partial z \partial y} & \frac{\partial^2 u}{\partial z^2} \end{bmatrix} \tag{187}$$

используется функция hessian(u, (x, y, z)). Далее в программе определяются частные производные функции u по ее аргументам: du_dx, du_dy, du_dz и для нахождения критических точек решается система уравнений

$$\begin{cases} \frac{\partial u}{\partial x}=0 \\ \frac{\partial u}{\partial y}=0 \\ \frac{\partial u}{\partial z}=0 \end{cases} \tag{188}$$

с помощью функции solve((du_dx, du_dy, du_dz), (x, y, z)).

Решения этой системы, возвращаемые данной функцией в виде списка кортежей и присваиваемые переменной solutions, имеют тип Rational. Для того, чтобы далее их корректно подставить в матрицу H, они преобразуются с помощью двойного цикла в список списков critical_points, элементы которого имеют тип float.

Затем в программе организуется цикл по всем критическим точкам, на каждой итерации которого координаты критических точек подставляются в матрицу Гессе посредством инструкции H_at_point = H.subs({x: point[0], y: point[1], z: point[2]})

и находятся собственные значения матрицы с помощью инструкции

eigenvalues = H_at_point.eigenvals().

Для того, чтобы сделать вывод о виде критической точки, вычисленные собственные значения в каждой точке сравниваются с нулем: если все собственные значения положительны, то критическая точка является минимумом; если все отрицательные - максимумом, а если знаки собственных чисел различны, то критическая точка является минимаксом или «седлом».

Вывод результатов работы программы:

```
Решения системы уравнений (тип Rational):
[(-1/4, -1/2, 1/4), (1/3, 2/3, -1/3)]
Критические точки (тип float):
[[-0.25, -0.5, 0.25], [0.3333333333333333, 0.6666666666666666, -0.3333333333333333]]
Матрица Гессе в точке [-0.25, -0.5, 0.25]:
⎡4    -1   2 ⎤
⎢            ⎥
⎢-1  -3.0  0 ⎥
⎢            ⎥
⎣2    0    2 ⎦
Собственные значения: {-3.15671458767231: 1, 5.32363965413612: 1, 0.833074933536192: 1}
Точка является седловой точкой.
Матрица Гессе в точке [0.3333333333333333, 0.6666666666666666, -0.3333333333333333]:
⎡4   -1   2 ⎤
```

⎢ ⎥
⎢-1 4.0 0⎥
⎢ ⎥
⎣2 0 2 ⎦

Собственные значения: {5.68133064360498: 1, 3.64207363248150: 1, 0.676595723913522: 1}

Точка является локальным минимумом.

Как видно, исследуемая функция имеет две критических точки:

[(-1/4, -1/2, 1/4), (1/3, 2/3, -1/3)],

из которых первая не является точкой экстремума, а является седловой точкой. Вторая точка является точкой экстремума - минимума.

Практическое применение аналитические методы оптимизации находят при исследовании уравнения регрессии второго порядка от двух переменных:

$$y = b_0 + b_1 x_1 + b_2 x_2 + b_{11} x_1^2 + b_{22} x_2^2 + b_{12} x_1 x_2 \qquad (189)$$

Необходимое условие экстремума (185) для данной функции принимает вид

$$\begin{aligned} b_1 + 2 b_{11} x_1 + b_{12} x_2 = 0 \\ b_2 + 2 b_{22} x_2 + b_{12} x_1 = 0 \end{aligned} \quad . \qquad (190)$$

Решая систему линейных уравнений (190) относительно неизвестных x_1, x_2 , находят координаты центра поверхности отклика x_1^0, x_2^0 :

$$x_1^0 = \frac{b_2 b_{12} - 2 b_1 b_{22}}{4 b_{11} b_{22} - b_{12}^2} \quad ,$$

$$x_2^0 = \frac{b_1 b_{12} - 2 b_2 b_{11}}{4 b_{11} b_{22} - b_{12}^2} \quad . \qquad (191)$$

При этом выясняется также, лежит ли центр поверхности внутри области допустимых значений независимых переменных (если в области физически допустимых значений x_1, x_2 функция изменяется монотонно, то центр поверхности будет лежать за пределами области допустимых значений x_1, x_2). Если центр поверхности находится внутри исследуемой области, то, подставив в уравнение найденные значения x_1^0, x_2^0 , можно определить значение целевой функции в точке экстремума.

Вид поверхности отклика, а вместе с ним и характер экстремума в центре поверхности можно определить на основании достаточного условия экстремума, которое для данного случая сводится к исследованию корней характеристического полинома матрицы квадратичной формы, связанной с уравнением регрессии. Корни характеристического многочлена находятся из решения относительно λ уравнения

$$\det(H-\lambda E)=0 \quad , \tag{192}$$

где E - единичная матрица, H - матрица Гессе квадратичной формы

$$H=\begin{pmatrix} 2b_{11} & b_{12} \\ b_{12} & 2b_{22} \end{pmatrix} \quad , \tag{193}$$

которая в данном случае является не зависящей от переменных x_1, x_2 .

Запишем уравнение (192) в явном виде

$$\begin{vmatrix} 2b_{11}-\lambda & b_{12} \\ b_{12} & 2b_{22}-\lambda \end{vmatrix}=0 \quad , \tag{194}$$

или

$$(2b_{11}-\lambda)(2b_{22}-\lambda)-b_{12}^2=0 \quad . \qquad (195)$$

Решая квадратное уравнение (195) относительно λ, находят собственные числа λ_1 и λ_2, соотношение знаков которых определяет вид поверхности отклика и вид экстремума. Функция отклика имеет вид эллиптического параболоида и экстремум в центре, если λ_1 и λ_2 имеют одинаковые знаки. При этом если значения λ_1 и λ_2 положительные, то в центре поверхности находится минимум, если отрицательные - то максимум. Если знаки λ_1 и λ_2 разные, то функция отклика имеет вид гиперболического параболоида, а в центре поверхности имеется "седло" (минимакс).

Пример 2. Исследовать график поверхности уравнения регрессии второго порядка при различных значениях коэффициентов.

Ниже приведен текст программы для исследования уравнения регрессии второго порядка на экстремум.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
#Задаем значения коэффициентов регрессии
b=[1,1,1,3,3,1]
#Определим функцию регрессии
def y(x1,x2):
   return(b[0]+b[1]*x1+b[2]*x2+b[3]*x1**2+b[4]*x2**2+b[5]*x1*x2)
#Определим матрицу Гессе и вектор линейных коэффициентов уравнения регрессии
G=np.array([[2*b[3],b[5]],[b[5],2*b[4]]])
b_G=np.array([-b[1],-b[2]])
print("Матрица Гессе:")
```

```
print(G)
#Вычислим координаты критической точки и значения функции регрессии в ней
x_extr=np.linalg.inv(G) @ b_G.T
print("x_extr:")
print(x_extr)
print("Значение отклика в точке x_extr: ", y(x_extr[0],x_extr[1]) )
print("Собственные значения матрицы Гессе:")
eigs=np.linalg.eig(G)[0]
print(eigs)
#Установим вид поверхности
if eigs[0]<0 and eigs[1]<0:
    print("Вид поверхности отклика: эллиптический параболоид, в центре поверхности - максимум. \n")
elif eigs[0]>0 and eigs[1]>0:
    print("Вид поверхности отклика: эллиптический параболоид, в центре поверхности - минимум. \n")
else:
    print("Вид поверхности отклика: гиперболический параболоид, в центре поверхности - седловая точка. \n")
#Строим график поверхности
x1 = np.linspace(-1, 1, 30)
x2 = np.linspace(-1, 1, 30)
X, Y = np.meshgrid(x1, x2)
Z = y(X, Y)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')  # Используем add_subplot для создания 3D-осей
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis')
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('Y(x1,x2)')
plt.show()
```

В начале программы задаются значений коэффициентов регрессии и определяется функция регрессии по уравнению, аналогичному (189). Далее определяется матрица Гессе (193), вектор линейных коэффициентов уравнения регрессии и вычисляются координаты критической точки путем решения системы уравнений (190) с помощью инструкции

```
x_extr=np.linalg.inv(G) @ b_G.T
```

Собственные значения матрицы Гессе рассчитываются с использованием функции np.linalg.eig(G)[0]. Вид поверхности функции уравнения регрессии определяется на основе анализа знаков собственных значений матрицы Гессе. Для значений коэффициентов регрессии, приведенных в тексте программы, она возвращает результат:

```
Матрица Гессе:
[[6 1]
 [1 6]]
x_extr:
[-0.14285714 -0.14285714]
Значения отклика в точке x_extr:  0.8571428571428572
Собственные значения матрицы Гессе:
[7. 5.]
Вид поверхности отклика: эллиптический параболоид, в центре поверхности - минимум.
```

График поверхности функции регрессии приведен на рисунке 16.

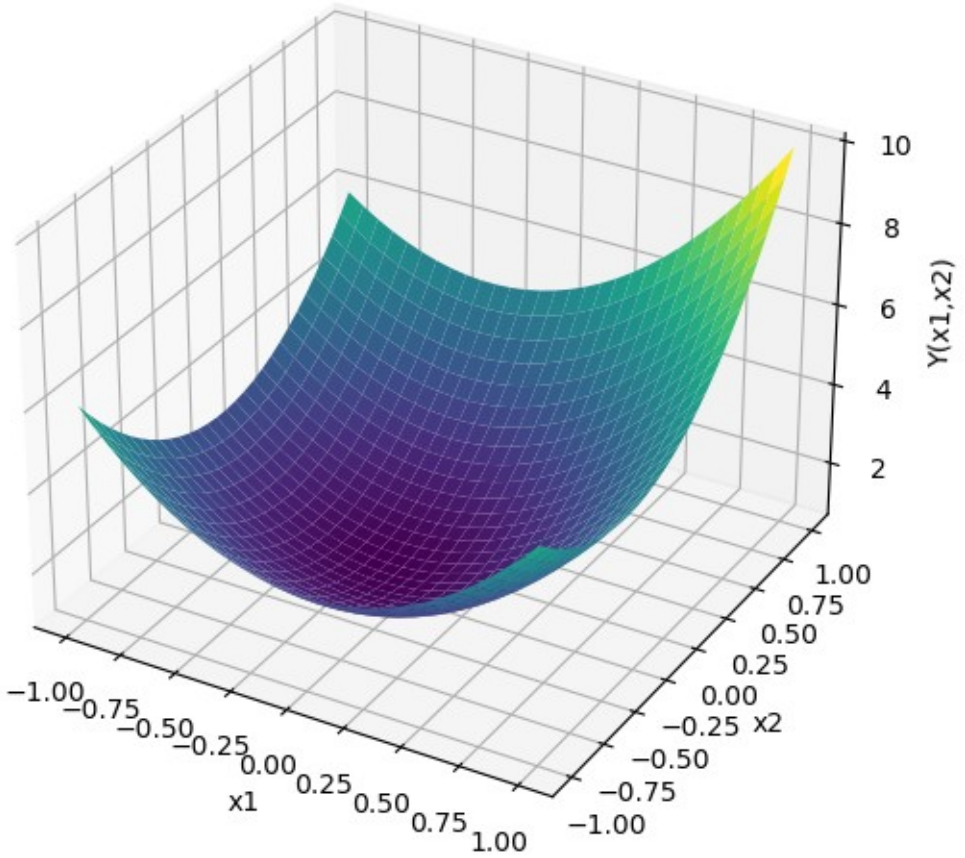

Рисунок 16. График поверхности функции регрессии при значениях коэффициентов: b=[1,1,1,3,3,1].

Для значений коэффициентов регрессии b=[1,1,1,-3,-3,1] результат работы программы следующий:

Матрица Гессе:

[[-6 1]

 [1 -6]]

x_extr:

[0.2 0.2]

Значение отклика в точке x_extr: 1.2

Собственные значения матрицы Гессе:

[-5. -7.]

Вид поверхности отклика: эллиптический параболоид, в центре поверхности - максимум.

График данной поверхности показан на рисунке 17.

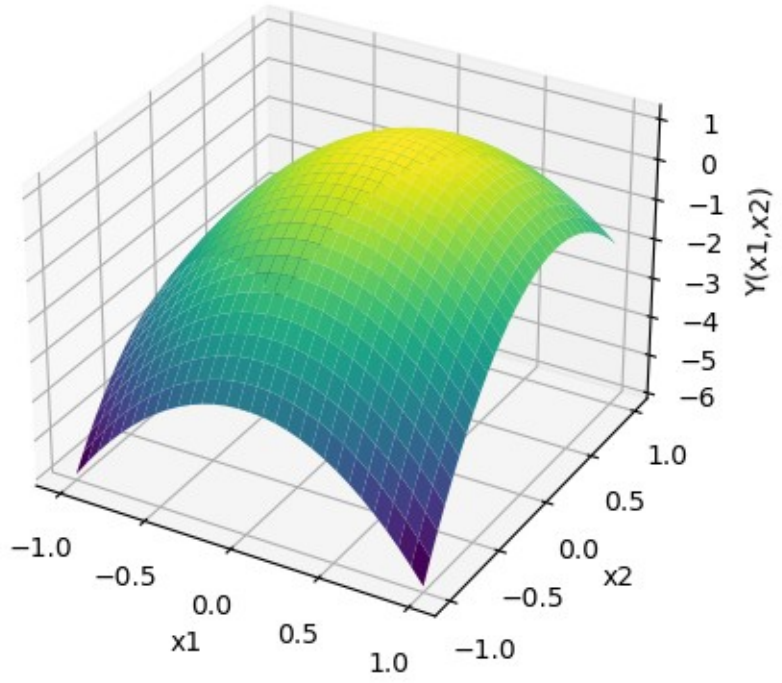

Рисунок 17. График поверхности функции регрессии при значениях коэффициентов: b=[1,1,1,-3,-3,1].

Для значений коэффициентов b=[1,1,1,3,-3,1] результат получается следующий:

Матрица Гессе:

[[6 1]

 [1 -6]]

x_extr:

[-0.18918919 0.13513514]

Значение отклика в точке x_extr: 0.9729729729729729

Собственные значения матрицы Гессе:

[6.08276253 -6.08276253]

Вид поверхности отклика: гиперболический параболоид, в центре поверхности - седловая точка.

В этом случае критическая точка представляет собой минимакс (рис. 18).

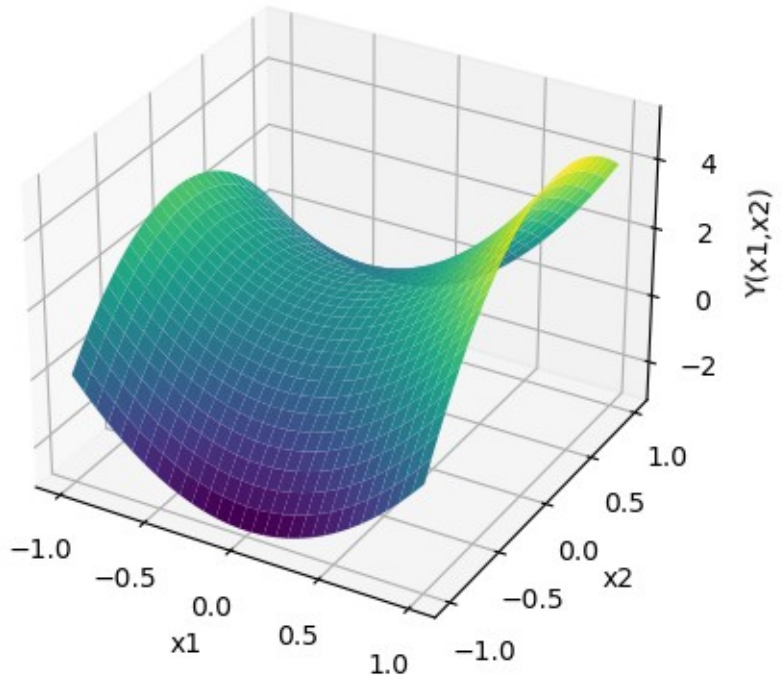

Рисунок 18. График поверхности функции регрессии при значениях коэффициентов: b=[1,1,1,3,-3,1].

Рассмотрим задачу об условной оптимизации при ограничениях вида равенств. В данном случае аналитически можно сформулировать условия экстремума. Для этого используется метод неопределенных множителей Лагранжа.

Пусть даны дважды непрерывно дифференцируемые целевая функция $f(x)=f(x_1,\dots,x_n)$ и функции ограничений $g_j(x)=g_j(x_1,\dots,x_n)=0, j=1..m, m<n$, определяющих множество допустимых решений X: $X=\{x|g_j=0, j=1..m, m<n\}$.

Требуется исследовать функцию $f(x)$ на экстремум, т. е. определить точки $x^0\in X$ ее локальных максимумов и минимумов на множестве X:

$$f(x^0)=\max_{x\in X} f(x),\ f(x^0)=\min_{x\in X} f(x) .$$

Определение. Функция

$$L(x,\lambda)=f(x)+\sum_{j=1}^{m}\lambda_j g_j(x) \qquad (196)$$

называется функцией Лагранжа, числа $\lambda_j, \lambda=(\lambda_1,\dots,\lambda_m)$ - множителями Лагранжа.

Теорема. (Необходимое условие экстремума при ограничениях типа равенств).

Пусть $x^0\in X$ - точка локального экстремума функции $f(x)=f(x_1,\dots,x_n)$ при ограничениях $g_j(x)=g_j(x_1,\dots,x_n)=0, j=1..m, m<n$.

Тогда найдутся такие числа $\lambda_j^0, \lambda^0=(\lambda_1^0,\dots,\lambda_m^0)$, не равные одновременно нулю, что выполняются следующие условия:

1) условие стационарности функции Лагранжа

$$\frac{\partial L(x^0,\lambda^0)}{\partial x_i}=0,\ \ i=1..n \ , \qquad (197)$$

2) условие допустимости решения

$$g_j(x^0)=g_j(x_1^0,\dots,x_n)^0=0, j=1..m \ .$$

Теорема. (Достаточное условие экстремума при ограничениях типа равенств).

Пусть имеется точка (x^0,λ^0) , в которой удовлетворяются требования 1) и 2) необходимого условия экстремума функции $f(x)=f(x_1,\dots,x_n)$ при ограничениях $g_j(x)=g_j(x_1,\dots,x_n)=0, j=1..m, m<n$.

Если в этой точке $d^2L(x^0,\lambda^0)<0$ $\left(d^2L(x^0,\lambda^0)>0\right)$ для всех ненулевых $dx\in R^n$ таких, что

$$dg_j(x^0)=\sum_{i=1}^{n}\frac{\partial g_j(x^0)}{\partial x_i}dx_i=0, j=1,\dots,m \ ,$$

то точка x^0 является точкой локального максимума (минимума).

На основании сформулированных выше теорем можно рекомендовать следующий алгоритм поиска экстремума функции при ограничениях вида равенств.

1) Составить функцию Лагранжа (196);

2) Записать необходимые условия экстремума (197);

3) Решить полученную систему относительно x, λ ;

4) Для найденных точек проверить достаточные условия экстремума:

   1. Записать выражение второго дифференциала функции Лагранжа $d^2 L(x, \lambda)$ в точке $(x^{0,} \lambda^0)$.

   2. Записать систему уравнений $dg_j(x^0) = \sum_{i=1}^{n} \frac{\partial g_j(x^0)}{\partial x_i} dx_i = 0, j = 1, \ldots, m$.

   3. Выразить из данной системы любые m дифференциалов dx_i через остальные $(n-m)$ и подставить в $d^2 L(x, \lambda)$.

   4. Установить определенность квадратичной формы $d^2 L(x^0, \lambda^0) < 0$ $\left(d^2 L(x^0, \lambda^0) > 0\right)$.

Пример 3. Найти экстремум функции $f(x) = x_1^2 + 2x_2^2 + 3x_3^2$ на множестве $X = \{x | x_1 + x_2 + x_3 - 2 = 0\}$.

Заметим, что в отсутствие ограничения функция имеет глобальный минимум в точке с координатами (0,0,0). Для того чтобы установить, имеет ли данная функция уловный экстремум при заданном ограничении, выполним вычисления по описанному выше алгоритму с использованием библиотеки sympy.

```
from sympy import *
# Определяем переменные
x1, x2, x3, lam, dx1, dx2, dx3, dlam = symbols('x1 x2 x3 lam dx1 dx2 dx3 dlam')
# Определяем функцию и ограничение
f = x1**2 + 2*x2**2 + 3*x3**2
```

```
g = x1 + x2 + x3-2

# Составим функцию Лагранжа:

L = f+lam*g

# Находим частные производные

dL_dx1 = diff(L, x1)

dL_dx2 = diff(L, x2)

dL_dx3 = diff(L, x3)

print(f"Частные производные функции Лагранжа:\ndL_dx1={dL_dx1}, \ndL_dx2={dL_dx2},\ndL_dx3={dL_dx3}")

# Находим критические точки, решая систему уравнений необходимого условия экстремума функции Лагранжа:

solutions = solve((dL_dx1, dL_dx2, dL_dx3, g), (x1, x2, x3,lam))

print(f"Решения системы уравнений: \n{solutions}")

# Определим второй дифференциал функции Лагранжа:

d2L=diff(L,x1,2)*dx1**2+diff(L,x2,2)*dx2**2+diff(L,x3,2)*dx3**2+diff(L,x3,2)*dlam**2+2*diff(L,x1,1,x2,1)*dx1*dx2+2*diff(L,x1,1,x3,1)*dx1*dx3+2*diff(L,x2,1,x3,1)*dx2*dx3

#Определим первый дифференциал функции ограничений:

dg=diff(g,x1)*dx1+diff(g,x2)*dx2+diff(g,x3)*dx3

#Подставим во второй дифференциал функции Лагранжа выражение dx1 через dx2 и dx3

d2L_sb=d2L.subs({dx1:-dx2-dx3})

print(f"Квадратичная форма функции Лагранжа: \n{d2L_sb}")Результат выполнения программы:
```

Решение:

Частные производные функции Лагранжа:

dL_dx1=lam + 2*x1,

dL_dx2=lam + 4*x2,

dL_dx3=lam + 6*x3

Решения системы уравнений:

{x1: 12/11, x2: 6/11, x3: 4/11, lam: -24/11}

Квадратичная форма функции Лагранжа:

6*dlam**2 + 4*dx2**2 + 6*dx3**2 + 2*(-dx2 - dx3)**2

Как видно, полученное выражение представляет собой положительно определенную квадратичную форму. Таким образом, критическая точка $\left(\frac{12}{11}, \frac{6}{11}, \frac{4}{11}\right)$ является точкой условного минимума заданной функции. Координаты условного минимума не совпадают с координатами точки глобального минимума.

4.3. Численные методы нахождения экстремума

Поскольку аналитическое решение задачи оптимизации возможно только для некоторых достаточно простых моделей, основное практическое значение имеют численные методы отыскания экстремума. Большая часть численных методов подразумевает решение задачи оптимизации на ЭВМ. Вместе с тем некоторые из них могут быть реализованы в виде постановки оптимального эксперимента. Это относится в первую очередь к пошаговым методам поиска экстремума.

В варианте постановки оптимального эксперимента (апостериорного планирования) последовательность поиска выглядит следующим образом. Сначала изучают характер поверхности отклика в районе первоначально выбранной точки факторного пространства с помощью специально спланированных "пробных" опытов. На основании пробных опытов определяют направление наиболее желательного изменения функции отклика. Затем совершают рабочее движение в сторону экстремума, причем это направление корректируется в процессе движения. После выхода в район экстремума, условием которого служит ухудшение отклика в любом направлении при выбранной величине шага, оптимальную точку можно уточнить одним из следующих способов: 1) постановкой дополнительных, особым образом спланированных опытов; 2) получением математической модели второго или более высокого порядка и последующим аналитическим решением задачи оптимизации для этой модели.

Общая особенность численных методов оптимизации состоит в том, что отыскание экстремума в них осуществляется путем последовательных приближений или итераций. При этом в каждом последующем приближении тем или иным способом учитывается информация, полученная в результате предшествующего движения. Поскольку функция отклика анализируется или после каждого шага, или после небольшого числа шагов, то естественным образом можно решать как задачу безусловной оптимизации, так и задачу отыскания компромиссного экстремума. В последнем случае в алгоритм поиска необходимо добавить проверку выполнения функций

ограничений, и если на каком-либо из шагов эти ограничения нарушатся, то соответствующим образом скорректировать движение.

В дальнейшем при рассмотрении различных численных методов для определенности будем считать, что необходимо отыскать максимум функции $y=f(x_1,..,x_n)$. В противном случае можно положить $y'=-y$ и искать максимум функции y' .

Общая итерационная последовательность поиска экстремума $f(\mathring{x})\approx \max\limits_{x\in\mathbb{R}^n} f(x)$ может быть сформулирована следующим образом:

1. Выбор начальной точки поиска x^0 .
2. Построение итерационной последовательности точек $\{x^k\}$, обладающих свойством $f(x^{k+1})>f(x^k), k=0,1,...$. Общее правило построения последовательности: $x^{k+1}=x^k+t_k d^k$, где d^k - направление движения, t_k - шаг.
3. Определение критерия окончания поиска.

В зависимости от способа вычисления d^k выделяют следующие три группы методов :

а) методы нулевого порядка — использование непосредственно значения функции $f(x^k)$,

б) методы первого порядка — использование первой производной функции $f'(x^k)$,

в) методы второго порядка — использование второй производной функции $f''(x^k)$.

4.3.1. Методы покоординатного движения к экстремуму

Данные методы предусматривают поочередное нахождение частных экстремумов целевой функции по каждому фактору $x_i(i=1,...,k)$. Алгоритм поиска состоит в следующем. Выбирают некоторую начальную точку поиска $x^0(x_1^0,...,x_k^0)$ и в этой точке фиксируют все независимые переменные кроме одной, например x_1 . При этом функция отклика становится функцией только одной переменной $f(x_1)$. Отыскивают максимум данной функции по

аргументу x_1 . В этой точке фиксируют значение данной переменной и начинают изменять следующую переменную x_2 . Отыскивают максимум функции $f(x_2)$ и фиксируют найденное значение x_2 . Этот процесс продолжают далее, пока не будет пройден цикл по всем переменным. Подобные циклы повторяют до тех пор, пока не будет достигнуто условие окончания поиска.

Разные методы покоординатного движения к экстремуму отличаются между собой способом отыскания экстремума функции одного переменного $f(x_i)$. Общая схема алгоритмов одномерной оптимизации может быть сформулирована следующим образом.

1. Выбор интервала неопределенности — отрезка, на котором функция предположительно имеет минимум.

2. Уменьшение интервала неопределенности на основе вычисления значений функции во внутренних точках интервала.

3. Проверка условия окончания поиска: длина текущего интервала неопределенности меньше установленного значения.

Среди различных алгоритмов уменьшения интервала неопределенности можно выделить следующие:

а) метод деления отрезка пополам $c_k = \frac{a_k + b_k}{2}$,

б) метод «золотого сечения» $c_k = a_k + \frac{3-\sqrt{5}}{2}(b_k - a_k)$.

в) метод Фибоначчи $c_k = a_k + \frac{F_{N-1}}{F_N}(b_k - a_k)$.

Здесь посредством a_k и b_k обозначены соответственно левая и правая границы интервала неопределенности, c_k - новая граница, F_{N-1}, F_N - числа Фибоначчи.

Рассмотрим более подробно алгоритм одномерной оптимизации методом деления отрезка пополам. Он основан на последовательном делении отрезка поиска пополам и выборе той его половины, на которой функция принимает большие значения. Алгоритм состоит из следующих шагов.

1. Задание начального отрезка поиска $[a,b]$.

Пусть функция $f(x)$ задана на отрезке $[a,b]$, где $a<b$. Это начальный отрезок поиска максимума функции.

2. Вычисление значений функции в точках деления отрезка.

Разделим отрезок $[a,b]$ пополам в точке $x=(a+b)/2$. Вычислим значения функции в точках a , $(a+b)/2$ и b .

3. Выбор половины отрезка с большими значениями функции.

Сравним значения функции в трех точках и выберем ту половину отрезка $[a,(a+b)/2]$ или $[(a+b)/2,b]$, на которой функция принимает большие значения. Эта половина и будет новым отрезком поиска.

Шаг 4. Повторение алгоритма

Повторяем шаги 2 и 3, пока длина отрезка поиска не станет меньше заданной точности ε. Тогда максимум функции будет находиться в середине последнего отрезка с точностью ε.

5. Определение максимума функции.

Максимум функции $f(x)$ на отрезке $[a,b]$ с точностью ε находится в точке $x=(a+b)/2$, где a и b - координаты последнего отрезка поиска.

Ниже приведен текст программы для нахождения максимума функции на отрезке, значения которой моделируются с ошибкой в виде нормально распределенного случайного числа с нулевым математическим ожиданием и заданным стандартным отклонением.

```
import numpy as np
import matplotlib.pylab as plt
def f(x,sigma):
    # Пример функции, которую мы хотим максимизировать
```

```
    err = np.random.normal(0, sigma, 1)
    return -((x - 2) ** 2) + 1 + err  # Максимум в точке x=2
def bisection_method_max(a, b, tol, max_iter):
    """
    Метод деления отрезка пополам для нахождения максимума функции.
    :param a: Левая граница отрезка
    :param b: Правая граница отрезка
    :param tol: Допустимая погрешность
    :param max_iter: Максимальное количество итераций
    :return: Приближенное значение максимума
    """
    if f(a,sigma) < f(b,sigma):
        a, b = b, a  # Убедимся, что f(a) >= f(b)
    iters=0
    for i in range(max_iter):
        mid1 = a + (b - a) / 3
        mid2 = b - (b - a) / 3
        if f(mid1,sigma) > f(mid2,sigma):
            b = mid2
        else:
            a = mid1
        iters +=1
        if abs(b - a) < tol:
            break
    return ((a + b) / 2,iters)
# Параметры
a = 0  # Левая граница
b = 4  # Правая граница
tol = 1e-5  # Допустимая погрешность
max_iter = 100  # Максимальное количество итераций
# Вывод результатов и построение графиков
```

```
N_sp=20
x=np.linspace(a,b,N_sp)
def make_plot(sigma):
    maximum, iters = bisection_method_max(a, b, tol, max_iter)
    print(f"Количество итераций: {iters}")
    print(f"Приближенное значение максимума: {maximum}")
    print(f"Значение функции в максимуме: {f(maximum,sigma)}")
    y=[]
    for i in range(len(x)):
        y.append(f(x[i],sigma))
    plt.title(f'sigma={sigma}')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.plot(x,y,"x")
    plt.plot(x,f(x,0))
    plt.plot(maximum,f(maximum,sigma),"o")
plt.figure(figsize=(10, 6))
sigma=0.01
plt.subplot(1, 2, 1)
make_plot(sigma)
sigma=0.5
plt.subplot(1, 2, 2)
make_plot(sigma)
plt.show()
```

В данной программе определяется тестовая функция f(x,sigma), которая моделирует зависимость $y=-(x-2)^2+1+err$, где err - случайная величина с распределением $N(0,sigma)$. Теоретическое значение максимума данной функции равно 1 при $x=2$. Для численного решения задачи одномерной оптимизации методом деления отрезка пополам определена функция bisection_method_max(a, b, tol, max_iter), в которой реализован алгоритм, описанный выше. Вывод результатов вычислений и построения графиков значений функции и найденного приближенного значения экстремума осуществляется с помощью функции

make_plot(sigma), параметром которой является стандартное отклонения ошибки моделируемой функции. В заключительной части программы решается задача и строятся графики для двух значений параметра sigma.

Ниже приводится пример вывода результатов, а также графики теоретического значения функции, смоделированных значений и точки найденного экстремума (рис.).

```
sigma=0.01
Количество итераций: 32
Приближенное значение максимума: 2.1494218433022216
Значение функции в максимуме: [0.97334229]
sigma=0.5
Количество итераций: 32
Приближенное значение максимума: 2.001636723865275
Значение функции в максимуме: [1.16823328]
```

Как видно из результатов, данный алгоритм нулевого порядка устойчив к ошибкам измерения функции отклика, в отличие от методов первого и второго порядков, которые очевидно не работали бы при больших значениях sigma.

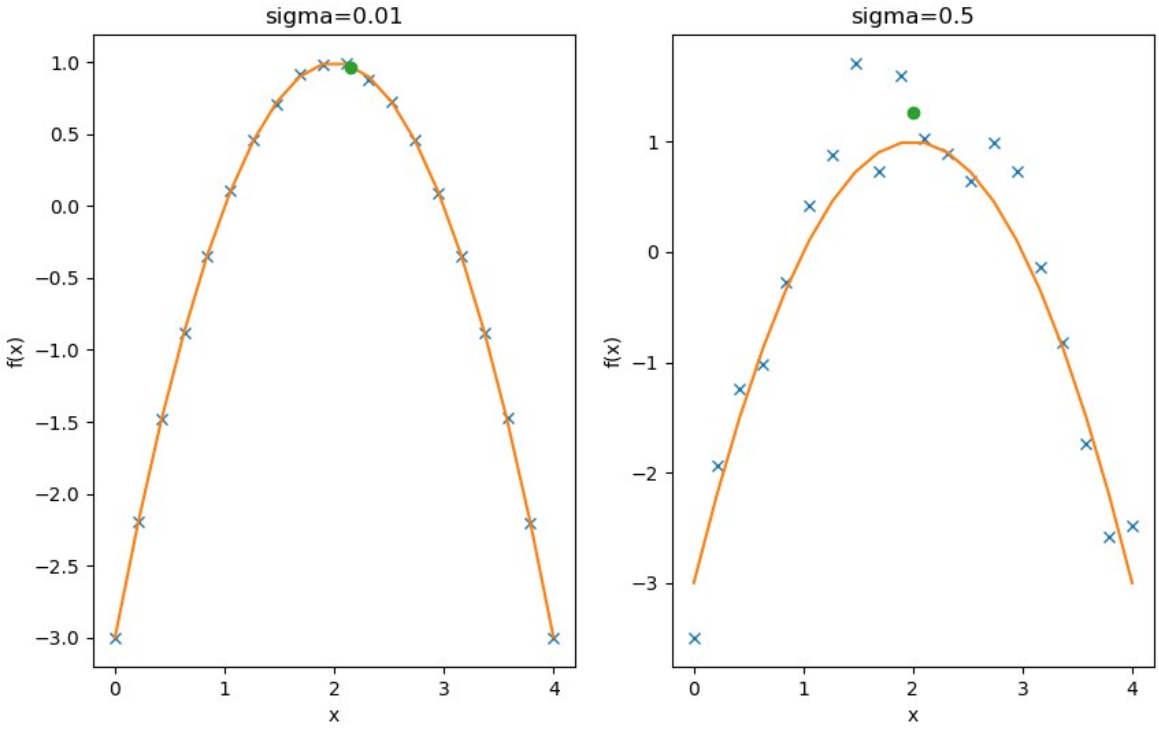

Рисунок 19. Иллюстрация результатов алгоритма одномерного поиска максимума функции методом деления отрезка пополам: кривая - теоретическое значение, "x"- смоделированные точки, "o"- найденный экстремум

Рассмотрим наиболее простой многомерный метод покоординатного поиска - метод Гаусса-Зейделя. Особенность его состоит в том, что отыскание экстремума по каждой переменной x_i осуществляется путем пошагового движения. Так, на первом этапе фиксируют все переменные, кроме x_1, для которой выбирают интервал варьирования Δx_1 и делают два пробных шага в противоположных направлениях, а именно: вычисляют функции $y_1 = f\left(x_1^0 - \Delta x_1\right)$ и $y_2 = f\left(x_1^0 + \Delta x_1\right)$. Большее из значений y_1, y_2 указывает направление движения к максимуму функции $f\left(x_1\right)$. Допустим, что большим оказалось y_2. Тогда в данном направлении осуществляют движение, вычисляя на каждом шаге значения $y^{k+1} = f\left(x_1^k + \Delta x_1\right)$.

Поиск экстремума по переменной x_1 прекращают, когда будет достигнуто условие $y^{k+1} < y^k$. В этом случае точку x_1^k принимают за точку максимума функции $f(x_1)$ и начинают изменять переменную x_2. Данный процесс продолжают по всем переменным до x_n. Затем цикл можно повторить, опять начав с переменной x_1. Поиск прекращают, когда шаг по каждой переменной будет приводить к уменьшению значения целевой функции. Точку, в которой это произойдет, можно принять за точку оптимума. В том случае, если требуется найти значение экстремума с большей точностью, необходимо уменьшить шаг и продолжить поиск, приняв точку экстремума за исходную.

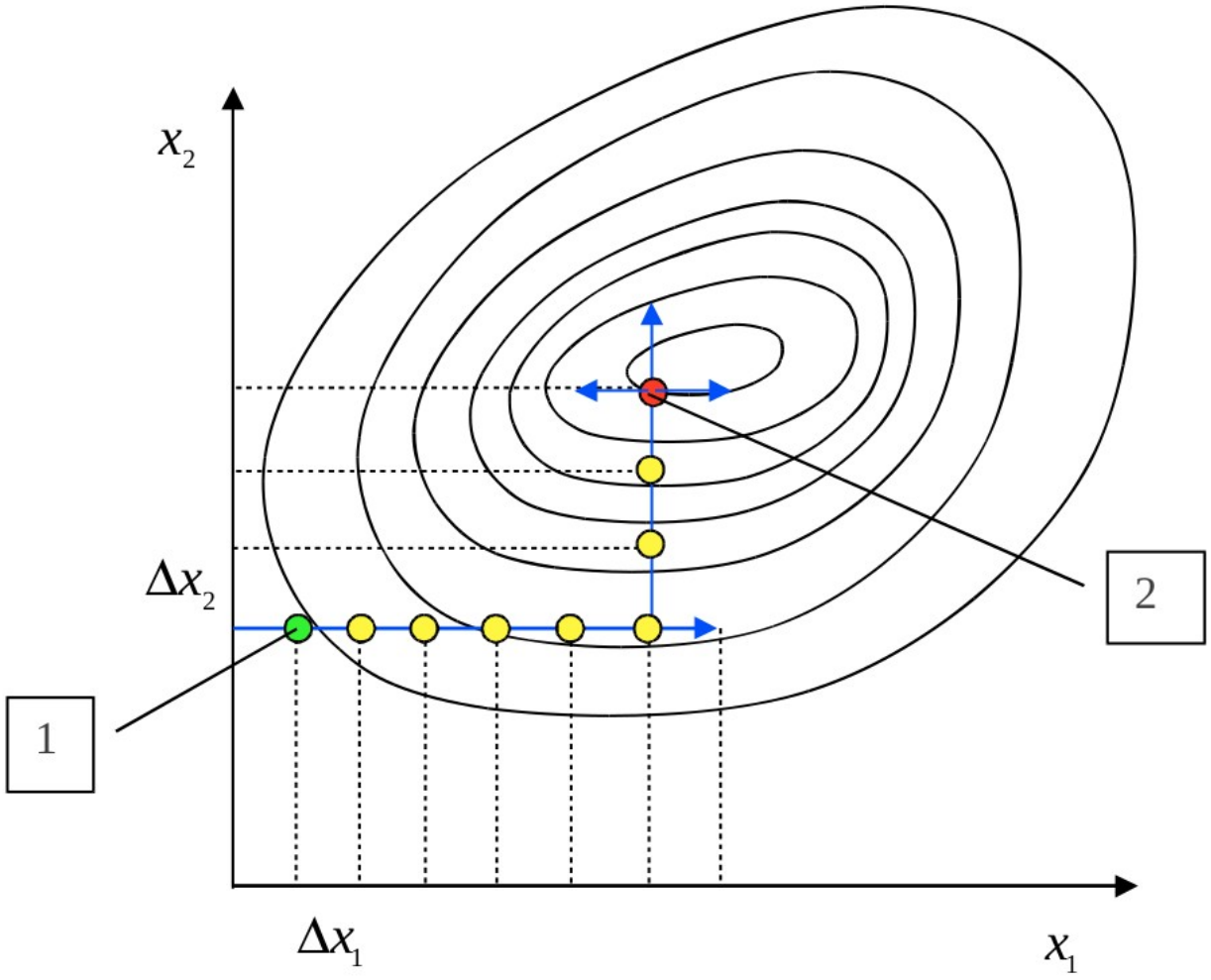

Рисунок 20. Схема движения к оптимуму в методе Гаусса-Зейделя

На рис. 20 представлена графическая иллюстрация к описанному алгоритму для двух переменных. Контурные линии соответствуют проекциям функции отклика $f(x_1, x_2)$ на плоскость независимых переменных $x_1 x_2$. Поиск начинается в точке 1 по переменной x_1 с шагом Δx_1 . После достижения экстремума по переменной x_1 движение продолжается по переменной x_2 с шагом Δx_2 . В точке 2 происходит окончание поиска, поскольку шаг в любом направлении будет приводить к уменьшению функции отклика.

Достоинства метода Гаусса-Зейделя - это высокая помехозащищенность и простота. Недостатком его является низкая скорость сходимости. Более эффективными методами нулевого порядка являются метод Бокса-Уилсона и симплексный поиск, которые применяются как при решении задачи оптимизации на ЭВМ, так и при постановке оптимального эксперимента непосредственно на физическом объекте исследования.

Заметим, что эффективность метода Гаусса-Зейделя можно повысить, если при движении по каждой координате реализовать один из алгоритмов одномерной оптимизации. Ниже приведена программа, в которой движение по каждой из координат осуществляется по алгоритму одномерного поиска.

```
import numpy as np
import matplotlib.pylab as plt
def f2(x1,x2,sigma):
   # Пример функции, которую мы хотим максимизировать
   err = np.random.normal(0, sigma, 1)
   return -((x1 - 1) ** 2+1.5*(x2 - 1) ** 2) + 1 + err  # Максимум в точке (1,1)
def bisection_method_max(x_ax,x_val,a, b, tol, max_iter):
   def f(x,sigma):
      if x_ax=="x1":
         return f2(x,x_val,sigma)
      elif x_ax=="x2":
         return f2(x_val,x,sigma)
   if f(a,sigma) < f(b,sigma):
```

```
        a, b = b, a  # Убедимся, что f(a) >= f(b)
    iters=[]
    for i in range(max_iter):
        mid1 = a + (b - a) / 3
        mid2 = b - (b - a) / 3
        if f(mid1,sigma) > f(mid2,sigma):
            b = mid2
        else:
            a = mid1
        mid_tmp=(a + b) / 2
        iters.append(mid_tmp) #Добавляем середину текущего отрезка в список
        if abs(b - a) < tol:
            break
    return ((a + b) / 2,iters)
# Параметры
tol = 1e-5  # Допустимая погрешность
max_iter = 100  # Максимальное количество итераций
sigma=0.01
#Ищем экстремум по оси x1
#Стартовый интервал по x1:
a = -3  # Левая граница по x1
b = 3  # Правая граница по x1
x2_0=-1 #Фиксированное значение x2
x1_opt, iters1 = bisection_method_max("x1",x2_0,a, b, tol, max_iter)
#Ищем экстремум по оси x2
#Стартовый интервал по x2
a = 1  # Левая граница по x2
b = 3  # Правая граница по x2
x2_opt, iters2 = bisection_method_max("x2",x1_opt,a, b, tol, max_iter)
print("Найденные координаты экстремума:")
print(f"x1_opt = {x1_opt:.4f}, количество итераций: {len(iters1)}")
```

```
print(f"x2_opt = {x2_opt:.4f}, количество итераций: {len(iters2)}")
x1 = np.linspace(-2, 3, 400)
x2 = np.linspace(-2, 3, 400)
X1, X2 = np.meshgrid(x1, x2)
# Вычисляем значения функции на сетке
Z = f2(X1, X2,0)
# Создаем контурный график и точки движения к экстремуму
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')
plt.colorbar(contour)
plt.title('Контурный график функции f(x)')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
x2_it=np.ones(len(iters1))*(-1)
x1_it=np.ones(len(iters2))*(x1_opt)
plt.scatter(iters1, x2_it, color='red', marker='x', label='Точки x2=const')
plt.scatter(x1_it, iters2, color='green', marker='x', label='Точки x1=const')
plt.scatter(x1_opt, x2_opt, color='blue', marker='o', label='Экстремум')
plt.legend()
plt.grid()
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.show()
```

В данной программе создается тестовая функция f2(x1,x2,sigma), которая моделирует зависимость отклика от двух переменных с ошибкой типа $N(0, sigma)$. Теоретическая точка максимума функции имеет координаты (1,1) со значением в данной точке равным 1. Для одномерного поиска по каждой независимой переменной используется функция bisection_method_max, модифицированная по сравнению с предыдущей программой таким образом, чтобы ее можно было использовать для оптимизации по двум переменным. С этой целью в нее добавлено два параметра x_ax, x_val. Первый параметр содержит символьную переменную, соотвествующую оси, по которой проводится поиск, а второй параметр - численное значение

второй координаты, фиксируемой во время движения по первой. Задавая нужные значения этим параметрам можно переходить последовательно с оптимизации по одной переменной к оптимизации по другой. Также внутри данной функции определяется еще одна функция f(x,sigma), которая в зависимости от значения символьного параметра x_ax позволяет фиксировать в функции отклика f2(x1,x2,sigma) одну или другую переменную и проводить одномерный поиск по оставшейся переменной. В данной программе в демонстрационных целях проведено по одному одномерному поиску по каждой из осей x1 и x2. Сначала фиксируется некоторое произвольное значение для оси x2 и находится условный максимум по оси x1. Потом фиксируется найденное значение x1_opt и проводится поиск по оси x2, в результате которого находится x2_opt. Найденные значения и количество итераций по каждой переменной выводятся на печать.

Результаты вычислений:

Найденные координаты экстремума:
x1_opt = 0.9446, количество итераций: 33
x2_opt = 1.0708, количество итераций: 31

В заключительной части программы строится контурный график теоретического значения целевой функции и на него наносятся точки, отвечающие движению к экстремуму по каждой переменной, а также точка найденного экстремума (рис. 21)

Как видно из результатов вычислений и графика, метод Гаусса-Зейделя с оптимизацией по каждой координате достаточно хорошо сходится даже при наличии случайных погрешностей в вычислении функции отклика. Точность алгоритма определяется величиной ошибки измерения функции отклика.

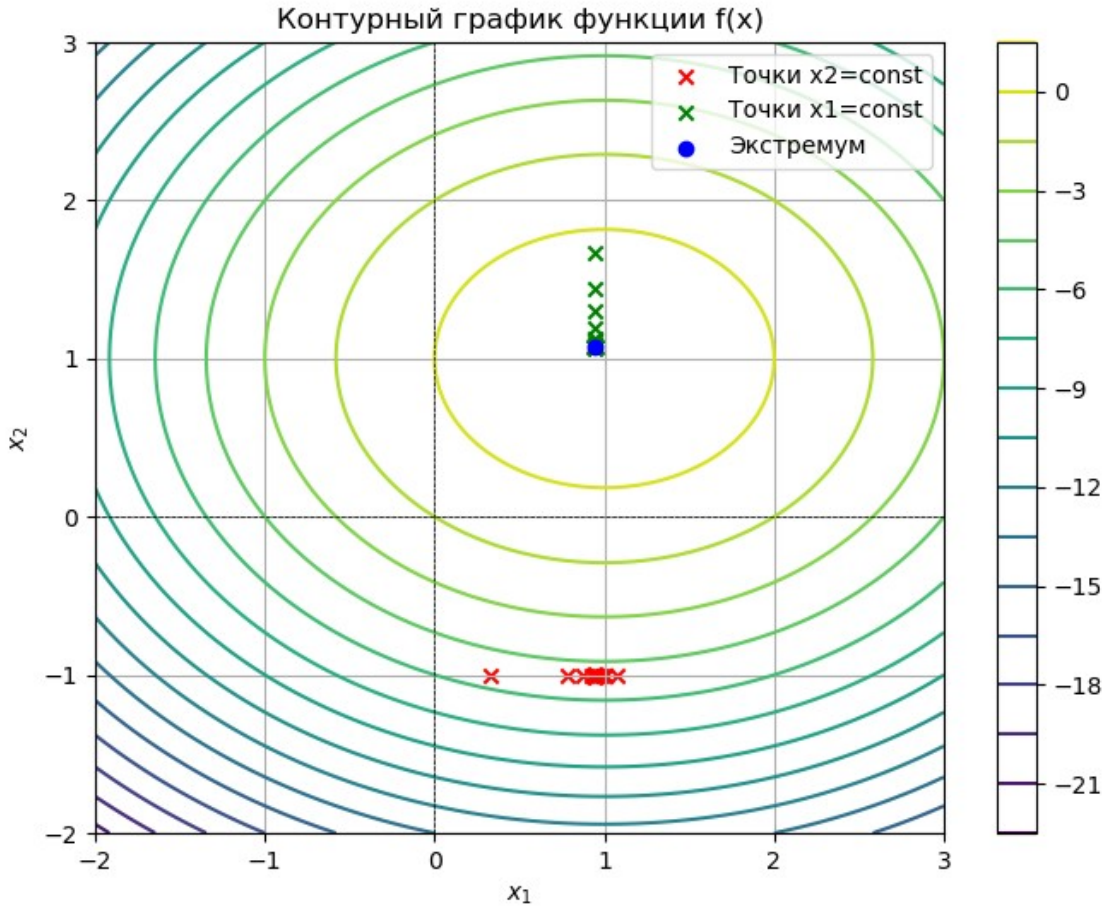

Рисунок 21. График, иллюстрирующий результаты работы программы поиска экстремума методом Гаусса-Зейделя с одномерной оптимизацией при движении по осям

4.3.2. Метод Бокса-Уилсона

Метод Бокса-Уилсона соединил в себе преимущества покоординатного движения к экстремуму и градиентных методов и таким образом является одним из наиболее подходящих методов для постановки оптимального эксперимента или апостериорного планирования эксперимента. Постановка оптимального эксперимента по существу является решением задачи оптимизации, с той лишь разницей, что целевая функция не вычисляется на основании математической модели, а определяется экспериментально непосредственно на объекте исследования. Логика поиска экстремума в методе Бокса-Уилсона в основном соответствует градиентному методу, поэтому в целом его можно считать разновидностью градиентных методов

оптимизации (подробно будут рассмотрены ниже). Особенность состоит в том, что градиент функции отклика вычисляют не дифференцированием самой функции отклика, а на основании линейной аппроксимации ее в локальной области. В направлении градиента линейной аппроксимации делается не один шаг, как обычно в градиентных методах, а несколько, причем на каждом шаге, так же как в методе Гаусса-Зейделя, проверяют условие возрастания самой функции отклика, а не вычисляют ее градиент. Это делает поиск более устойчивым к помехам, поскольку операция вычисления производной менее устойчива к случайным ошибкам, чем операция вычисления самой функции.

Рассмотрим более подробно алгоритм поиска на примере функции двух переменных. Пусть целевая функция y зависит от двух факторов x_1 и x_2, и ее значение может быть экспериментально определено при любом сочетании факторов из области допустимых значений. Последовательность поиска максимума функции y состоит из следующих шагов.

Выбирают начальную точку с координатами x_1^0, x_2^0 (в дальнейшем подразумевается, что уровни факторов пересчитываются в условный масштаб по формуле (152)), выбирают шаг варьирования для каждого фактора Δx_1, Δx_2 и в окрестности начальной точки реализуют планированный эксперимент по схеме линейного плана. Матрица планирования в нормированных единицах в данном случае включает 4 опыта и имеет следующий вид:

Таблица 13. Матрица планирования в методе Бокса-Уилсона

№ опыта	x_1	x_2	y
1	-1	-1	y_1
2	+1	-1	y_2
3	-1	+1	y_3
4	+1	+1	y_4

На основании проведенных четырех опытов рассчитывают коэффициенты уравнения линейной регрессии

$$y = b_0 + b_1 x_1 + b_2 x_2 \quad . \tag{198}$$

Данное уравнение является фактически аппроксимацией плоскостью функции отклика в локальной окрестности точки x^0 . В силу того, что матрица планирования линейного плана обладает свойством ортогональности, коэффициенты уравнения регрессии вычисляются по простым формулам

$$\begin{aligned} b_1 &= \frac{1}{4}(-y_1 + y_2 - y_3 + y_4) \\ b_2 &= \frac{1}{4}(-y_1 - y_2 + y_3 + y_4) \end{aligned} \quad . \tag{199}$$

Нас будут интересовать только коэффициенты b_1 и b_2 , поскольку именно они являются компонентами градиента линейного приближения (проекциями вектора градиента на координатные оси).

$$\begin{aligned} \frac{\partial y}{\partial x_1} &= b_1, \\ \frac{\partial y}{\partial x_2} &= b_2 \end{aligned} \quad . \tag{200}$$

Вычисленные коэффициенты b_1 и b_2 используются для расчета направления движения по градиенту линейного приближения, в соответствии с которым делают шаги в данном направлении, определяя экспериментально значения отклика в точках y^k , координаты которых вычисляют по формулам

$$\begin{aligned} x_1^k &= x_1^{k-1} + b_1 \Delta x_1 \\ x_2^k &= x_2^{k-1} + b_2 \Delta x_2 \end{aligned} \quad . \qquad (201)$$

На каждом шаге проверяют условие возрастания функции отклика

$$y^k > y^{k-1} \quad . \qquad (202)$$

В точке, где условие возрастания отклика нарушается, первый цикл движения к экстремуму считается законченным.

После окончания первого цикла можно поступить одним из следующих способов:

5. Если нет уверенности, что достигнутая точка близка к оптимуму, продолжить поиск. Для этого, считая найденную точку начальной, повторить поиск согласно вышеописанному алгоритму с тем же или с меньшим шагом.
6. Если имеются основания считать, что найденная в результате первого цикла точка является близкой к экстремуму, то линейного приближения для описания поверхности отклика в данной области будет недостаточно, и необходимо, используя какой-либо из планов второго порядка, описать целевую функцию в данной области уравнением параболической регрессии и затем исследовать его для нахождения максимума.

На рисунке 22 показана схема движения к оптимуму в методе Бокса-Уилсона. Начало поиска соответствует точке 1. Точки 2 отвечают координатам линейного плана 2^2, на основании которого вычисляются коэффициенты линейного уравнения регрессии, аппроксимирующего поверхность отклика вблизи начальной точки. Стрелкой указано направление градиента, вычисленное по результатам этих опытов. Далее движение производится по прямой, соответствующей направлению градиента линейного приближения. Движение заканчивается в точке 3, где функция отклика перестает возрастать. Как видно, эта точка не совпадает с точкой экстремума 4, поскольку в общем случае начальное направление градиента не соответствует в точности направлению на экстремум. В области, близкой к экстремуму, следует продолжить поиск одним из перечисленных выше способов.

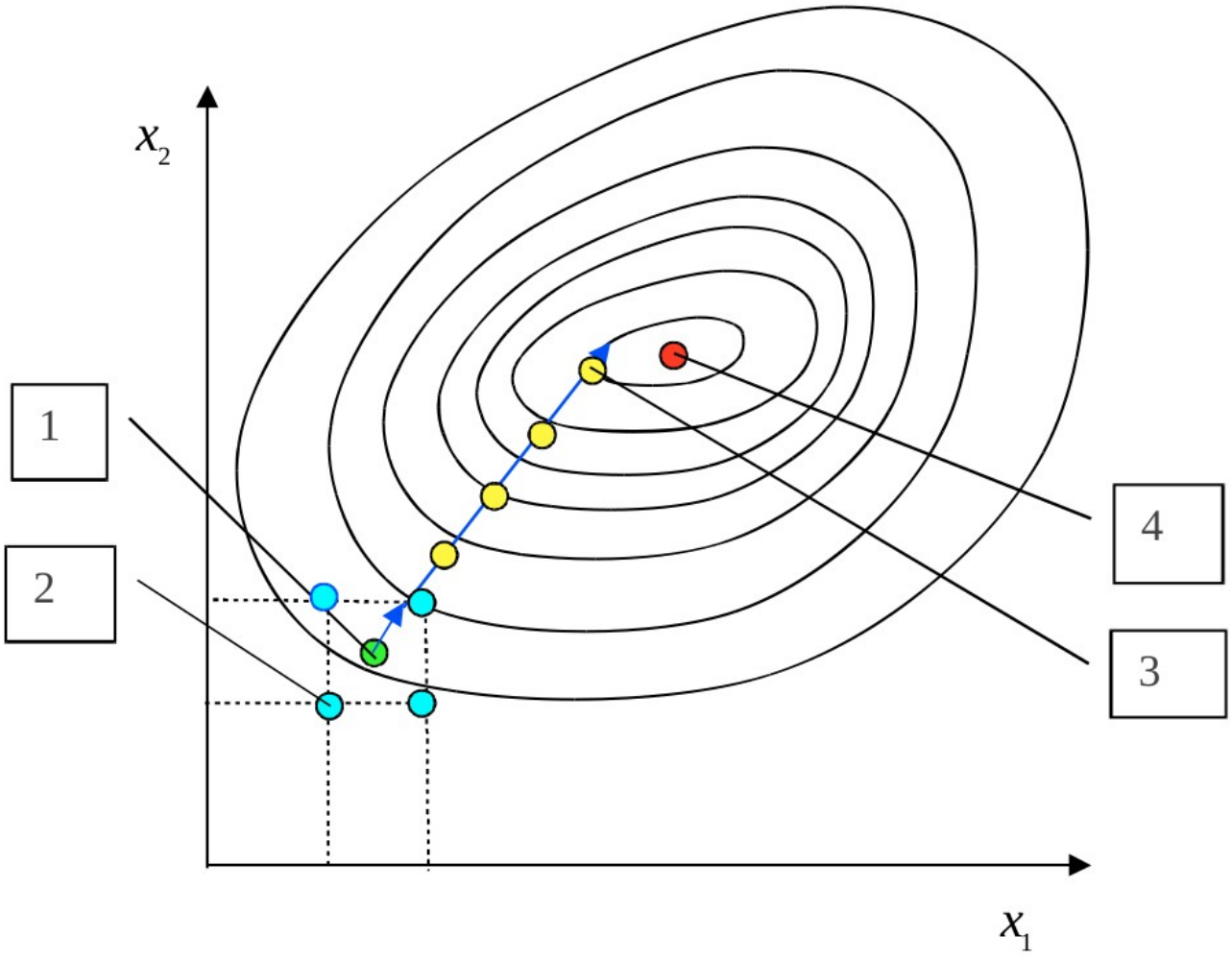

Рисунок 22. Схема движения к оптимуму в методе Бокса — Уилсона

Ниже приведен текст программы, в которой реализован метод Бокса-Уилсона применительно к оптимизации функции, рассмотренной в предыдущем примере.

```
import numpy as np
from numpy.linalg import inv
import matplotlib.pylab as plt
```

```
def f2(x1,x2,sigma):
    # Пример функции, которую мы хотим максимизировать
    err = np.random.normal(0, sigma)
    return -((x1 - 1) ** 2+1.5*(x2 - 1) ** 2) + 1 + err  # Максимум в точке (1,1)
# Параметры
max_iter = 100  # Максимальное количество итераций
sigma=0.01
# Функция для создания матрицы планирования
def make_X(x_0,dx):
    X=np.array([[1,x_0[0]-dx,x_0[1]-dx],
       [1,x_0[0]+dx,x_0[1]-dx],
       [1,x_0[0]-dx,x_0[1]+dx],
       [1,x_0[0]+dx,x_0[1]+dx]])
    return (X)
# Функция движения к экстремуму методом Бокса-Уилсона
def BW_1(x_0,dx):
    X=make_X(x_0,dx)
    y_e=np.zeros(4)
    for i in range(4):
        y_e[i]=f2(X[i,1],X[i,2],sigma)
    hat_b=inv(X.T @ X)@(X.T @ y_e)
    iters=[[x_0[0],x_0[1],f2(x_0[0],x_0[1],sigma)]]
    for i in range(1,max_iter):
        x1i=x_0[0]+hat_b[1]*i*dx
        x2i=x_0[1]+hat_b[2]*i*dx
        iters.append([x1i,x2i,f2(x1i,x2i,sigma)])
        if iters[i][2]<iters[i-1][2]: break
    return(iters[:-2])
#Первая серия итераций
x_0=[-1,-1] # Начальная точка
dx=0.1 # Шаг варьирования
```

```
X0=make_X(x_0,dx) # Исходная матрица планирования
iters1=BW_1(x_0,dx) # Первая серия итераций
x_0=iters1[-1][:2] # Новая начальная точка - последняя из первой серии
dx=dx/2 # Шаг варьирования уменьшаем
X01=make_X(x_0,dx) # Начальная матрица для второй серии итераций
iters2=BW_1(x_0,dx) # Вторая серия итераций
iters=iters1.copy()
iters.extend(iters2)
x1_opt,x2_opt,y_opt=iters[-1] # Точка, близкая к экстремуму
print("Найденные координаты экстремума:")
print(f"x1_opt = {x1_opt:.4f}, x2_opt = {x2_opt:.4f}, y_opt = {y_opt:.4f} ")
print(f"Количество итераций: {len(iters)}")
iters_arr=np.array(iters)
x1_it=iters_arr[:,0]
x2_it=iters_arr[:,1]
x1 = np.linspace(-2, 3, 400)
x2 = np.linspace(-2, 3, 400)
X1, X2 = np.meshgrid(x1, x2)
# Вычисляем значения функции на сетке
Z = f2(X1, X2,0)
# Создаем контурный график и точки движения к экстремуму
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')
plt.colorbar(contour)
plt.title('Контурный график функции f(x)')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.scatter(X0[:,1], X0[:,2], color='red', marker='x', label='Матрица X0')
plt.scatter(X01[:,1], X01[:,2], color='orange', marker='x', label='Матрица X1')
plt.scatter(x1_it, x2_it, color='green', marker='x', label='Точки итераций')
plt.scatter(x1_opt, x2_opt, color='blue', marker='o', label='Экстремум')
```

```
plt.legend()
plt.grid()
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.show()
```

В данной программе определяется функция make_X(x_0,dx) для создания матрицы планирования X (табл.13), которая используется затем в функции BW_1(x_0,dx), реализующей процедуру определения направления движения по градиенту линейного приближения и серии шагов в данном направлении в соответствии с алгоритмом Бокса-Уилсона. Ввиду того, что независимые переменные не переводятся в безразмерный масштаб, формулы (199) не используются, а коэффициенты уравнения регерссии (198) вычисляются при помощи общей формулы регрессионного анализа (159). Вычисленные коэффициенты уравнения регрессии используются для движения в направлении градиента в соответствии с формулами (201) до тех пор, пока выполняется условие возрастания функции отклика (202). После этого строится новая матрица планирования с меньшей величиной интервала варьирования переменных, определяется новое направление движения и выполняется следующая серия итераций. В данной программе делается всего две таких серии, что является в данном случае достаточным для нахождения точки, близкой к экстремуму, что иллюстрируется результатами вычислений и графиком движения к экстремуму (рис. 23).

Результат вычислений:

```
Найденные координаты экстремума:
x1_opt = 0.9397, x2_opt = 1.0598, y_opt = 0.9782
Количество итераций: 15
```

Количество итераций для нахождения координат экстремума с приблизительно такой же точностью, что и в методе Гаусса-Зейделя, в случае метода Бокса-Уилсона оказывается существенно меньшим, что говорит о его более высокой эффективности.

На практике после того, как найдена предполагаемая критическая точка в результате движения по градиенту линейного приближения, вместо того, чтобы проводить вторую серию опытов движения по градиенту, как сделано в настоящей программе, можно реализовать в окрестности критической точки планированный эксперимент по плану второго порядка, вычислить

коэффициенты уравнения регрессии второго порядка и исследовать его на экстремум с помощью аналитических методов.

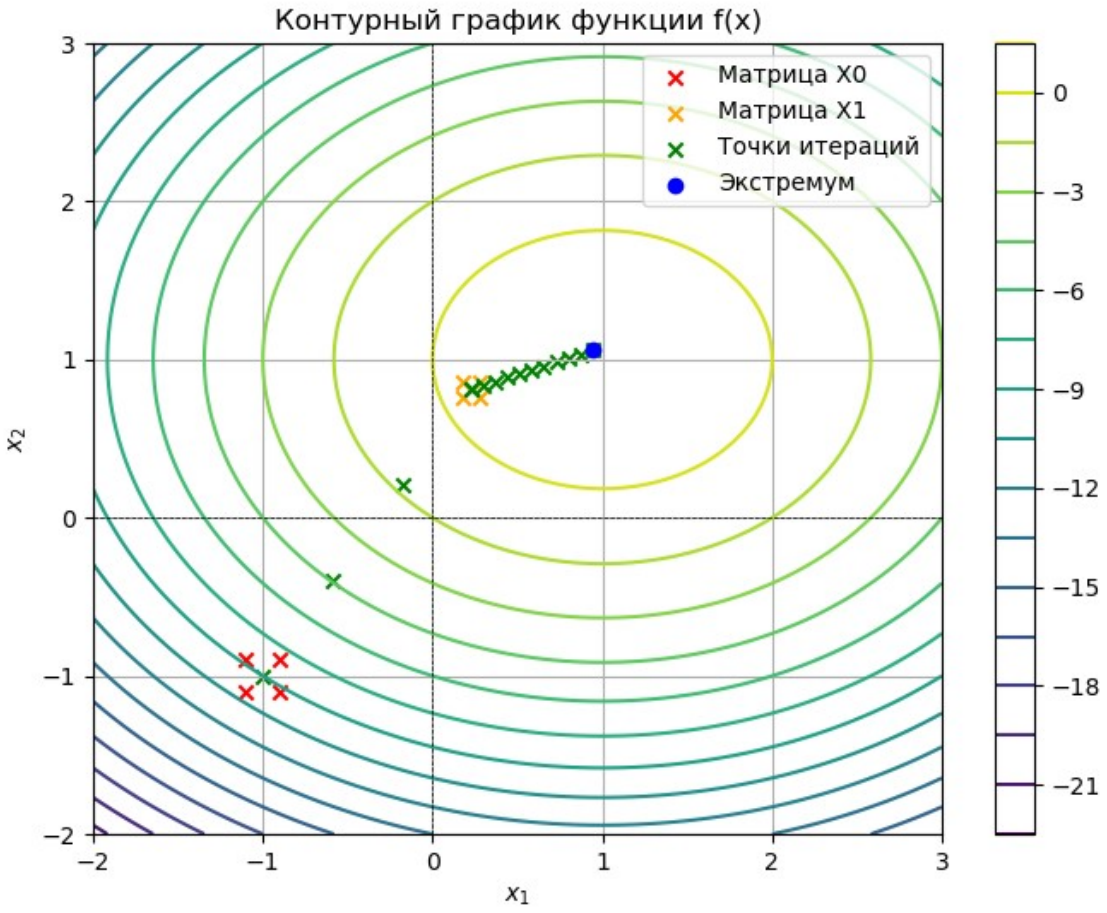

Рисунок 23. Целевая функция и точки движения к экстремуму методом Бокса-Уилсона

4.3.3. Симплексный поиск

Метод симплексного поиска может быть использован как алгоритм для постановки оптимального эксперимента, а также для решения задачи оптимизации на ЭВМ. По скорости сходимости он несколько уступает методу Бокса - Уилсона и другим градиентным методам, но значительно превышает их по надежности, поскольку в нем отсутствует чувствительная к ошибкам операция вычисления градиента. Кроме того, он обладает тем преимуществом, что в нем вообще не

требуется вычислять значение функции отклика - достаточно просто качественно оценивать результат по принципу "лучше - хуже". Это особенно важно, когда речь идет об оптимизации показателей, не поддающихся количественной оценке, например, технологичности процесса, или эстетических характеристик материала. В данном методе естественным образом также могут быть учтены ограничения, если ставится задача условной оптимизации.

В основе его лежит построение координат точек движения к оптимуму в вершинах регулярного многогранника в факторном пространстве - симплекса. Для двух факторов регулярный симплекс представляет собой равносторонний треугольник. Координаты вершин ***k***-мерного симплекса могут быть рассчитаны с помощью следующей матрицы:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ p_k & q_k & q_k & q_k \\ q_k & p_k & q_k & q_k \\ . & . & . & . \\ q_k & q_k & q_k & p_k \end{pmatrix} , \qquad (203)$$

где $p_k = \frac{1}{k\sqrt{2}}\left(\sqrt{k+1}+k-1\right)$, $q_k = p_k - \frac{\sqrt{2}}{2}$.

Строки этой матрицы представляют собой координаты вершин симплекса. Первая строка отвечает вершине, находящейся в начале координат.

Суть метода состоит в том, что опыты производят в точках с координатами вершин симплекса. То есть матрица (203) представляет собой начальную матрицу планирования. Для двух переменных она содержит три строки. Таким образом, чтобы начать движение, в данном методе достаточно трех опытов, тогда как в методе Бокса-Уилсона необходимо четыре. После того как первые опыты сделаны, необходимо сравнить отклики между собой и выбрать ту вершину симплекса, в которой отклик принимает наименее желательное значение (наихудшую точку). Дальнейшее движение производится путем отражения симплекса относительно наихудшей точки: наихудшая вершина отбрасывается и строится новый симплекс на противолежащей грани, зеркально расположенный относительно старого. Координаты новой вершины рассчитываются по следующей формуле:

$$x_{si}^{new} = \frac{2}{k}\left(x_{1i} + x_{2i} + \ldots + x_{s-1\,i} + x_{s+1\,i} + \ldots + x_{k+1\,i}\right) - x_{si} \quad , \qquad (204)$$

где x_{si} - i-я координата наихудшей вершины (s), x_{si}^{new} - эта же координата новой вершины, в скобках стоит сумма значений i-й координаты по всем остальным вершинам, кроме наихудшей.

Поскольку вершины представляют собой строки матрицы, а координаты - столбцы, то для того, чтобы построить новую строку, надо просто сложить все строки матрицы поэлементно, за исключением наихудшей, умножить на $2/k$ и вычесть из каждого элемента соответствующую ему координату наихудшей строки.

В этой новой точке необходимо произвести опыт и сравнить значение отклика со всеми остальными вершинами симплекса, выбрать в новом симплексе наихудшую точку и продолжать движение путем отражения симплекса относительно новой наихудшей точки. Движение заканчивают, когда симплекс начинает вращаться относительно какой-либо наилучшей точки. Ее координаты можно принять за центр области, близкой к экстремуму.

На рис. 24 показана схема движения к оптимуму симплексным методом для функции двух переменных. Исходный симплекс представляет собой правильный треугольник с одной из вершин в начале координат (точка 1). Как видно из рисунка, для данного примера именно эта точка является наихудшей. Следующая точка строится путем отражения точки 1 относительно противолежащей грани. При этом координаты новой точки вычисляются по формуле (204) для k=2. Фактически для получения строки с координатами новой точки в матрице (203) необходимо сложить элементы всех строк кроме наихудшей точки (строки 2 и 3) и вычесть строку наихудшей точки (строку 1). В получившемся новом симплексе выбирают новую наихудшую точку и продолжают движение аналогичным образом. Точка 3 отвечает концу поиска, поскольку дальнейшее движение приведет к "зацикливанию" симплекса - вращению его относительно точки оптимума 2.

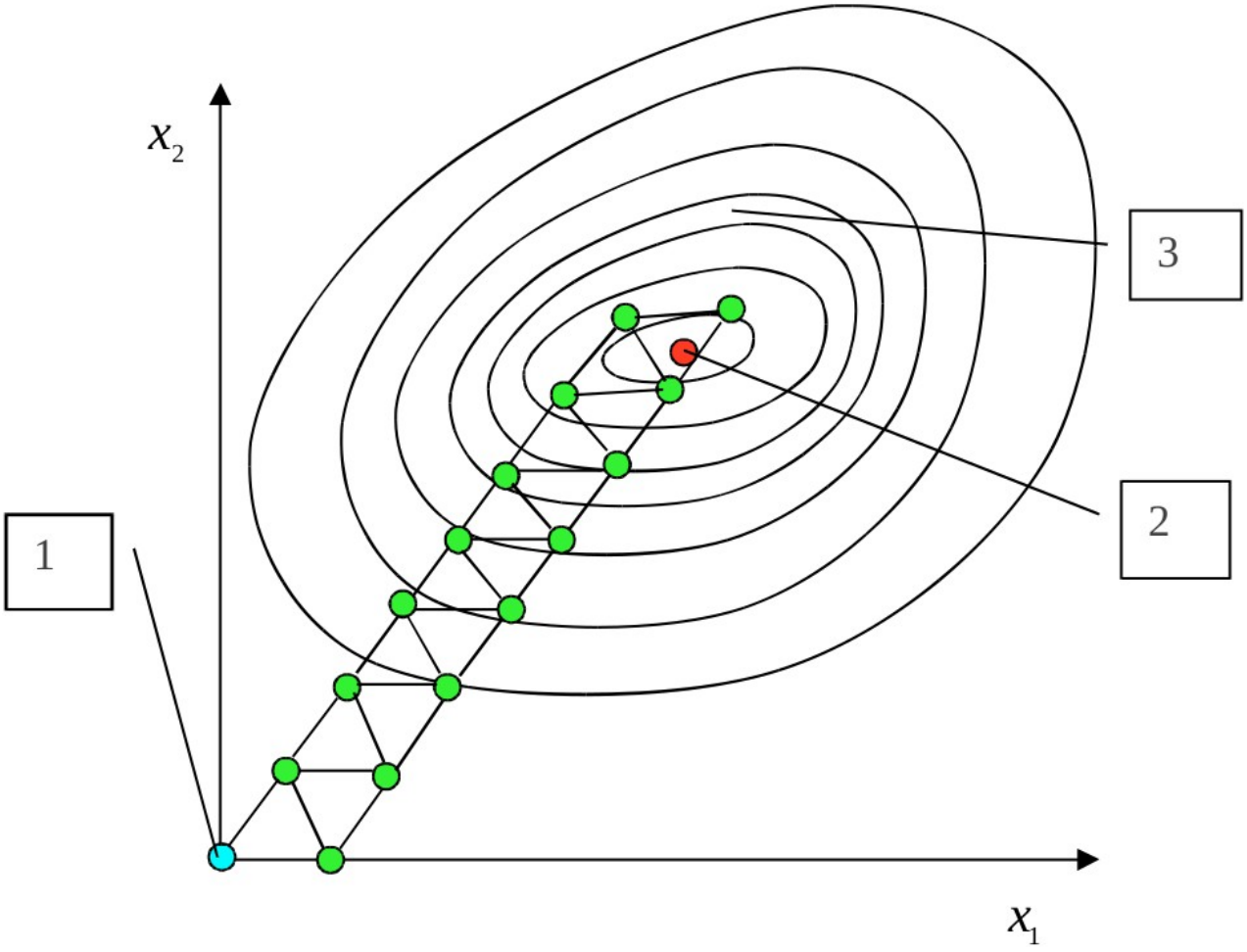

Рисунок 24. Схема движения к оптимуму симплексным методом: 1 - начальная точка, 2 - точка оптимума, 3 - точка окончания поиска

Описанный алгоритм отвечает самому простому варианту симплексного поиска. Существует и другие, более эффективные, варианты данного метода. Одним из наиболее популярных является метод Нелдера — Мида [36], особенностью которого является не только перемещение, но и деформация симплекса в процессе оптимизации. Данный метод входит в число методов многомерной минимизации библиотеки scipy.optimize. Ниже приведен пример программы, в которой используется функция из данной библиотеки.

```
import numpy as np
from scipy.optimize import minimize
def f2(x1, x2, sigma):
    err = np.random.normal(0, sigma)
```

```
    return -((x1 - 1) ** 2 + 1.5 * (x2 - 1) ** 2) + 1 + err  # Максимум в точке (1,1)
def objective_function(x, sigma):
    # Оборачиваем функцию для использования с minimize
    return -f2(x[0], x[1], sigma)  # Ищем минимум отрицательной функции
def nelder_mead(f, x0, sigma, tol=1e-6, max_iter=15):
    n = len(x0)
    simplex = np.zeros((n + 1, n))
    simplex[0] = x0
    for i in range(1, n + 1):
        x = simplex[0].copy()
        x[i - 1] += 1
        simplex[i] = x
    # Используем minimize с методом Нелдера-Мида
    res = minimize(f, simplex[0], method='nelder-mead',
                   options={'xatol': tol, 'maxiter': max_iter, 'initial_simplex': simplex})
    return res.x
# Пример использования
x0 = np.array([0, 0])  # Начальная точка
sigma = 0.01  # Стандартное отклонение ошибки
result = nelder_mead(lambda x: objective_function(x, sigma), x0, sigma)
print(f"Найденный максимум: x1_opt = {result[0]:.4f}, x2_opt = {result[1]:.4f}")
print(f"Значение функции в найденной точке: {f2(result[0], result[1], sigma):.4f}")
```

В данной программе определяется функция f2(x1, x2, sigma), аналогично предыдущим программам, на основе которой создается функция objective_function(x, sigma), используемая в дальнейшем при передаче в качестве параметра в функцию минимизации minimize. При этом у исходной функции меняется знак на противоположный, поскольку в функциях библиотеки scipy.optimize решается задача нахождения минимума, тогда как в нашей задаче требуется найти максимум заданной функции. Исходный симплекс строится с помощью функции nelder_mead. Функция minimize библиотеки scipy.optimize позволяет решать задачу оптимизации с использованием разных алгоритмов. Тип алгоритма оптимизации передается в числе параметров данной функции. В нашем примере задается параметр method='nelder-mead'. указывающий на то,

что при оптимизации должен использоваться алгоритм Нелдера-Мида. Кроме этого задаются также другие необходимые параметры, в том числе: целевая функция, координаты исходного симплекса, максимальное количество итераций 'maxiter': max_iter, которое в данном примере установлено равным 15. В конце программы на печать выводятся координаты найденного экстремума и значение целевой функции в точке экстремума.

Результат вычислений:

```
Найденный максимум: x1_opt = 1.0000, x2_opt = 1.0000
Значение функции в найденной точке: 0.9940
```

Как видно метод Нелдера-Мида достаточно эффективен, поскольку при том же числе итераций, что и в методе Бокса-Уилсона обеспечивает более высокую точность нахождения координат точки экстремума.

4.3.4. Градиентные методы

Градиентные методы имеют несколько разновидностей. Сущность их состоит в определении значений независимых переменных, дающих наибольшее изменение целевой функции. Это достигается при движении вдоль градиента функции. Разные методы отличаются друг от друга способом определения направления движения к оптимуму, выбором шага, продолжительностью поиска вдоль выбранного направления и критериями окончания поиска.

Для функции отклика $y=f(x_1,..,x_n)$ градиент в точке $x(x_1,..,x_n)$ представляет собой вектор с координатами

$$\nabla y=(\frac{\partial y}{\partial x_1},\ldots,\frac{\partial y}{\partial x_n}) \quad . \tag{205}$$

Рассмотрим простейший вариант градиентного метода. Последовательность операций при поиске оптимума сводится к следующему.

1. Выбирается начальная точка $x^0(x_1^0,\ldots,x_n^0)$, в которой вычисляется градиент целевой функции $\nabla f(x^0)$.

2. Делается шаг в направлении градиента и вычисляется новое значение целевой функции $f(x^1)=f(x^0+\alpha\nabla f(x^0))$, где α - положительная константа, определяющая величину шага.

3. В точке x^1 вычисляют градиент и делают следующий шаг $f(x^2)=f(x^1+\alpha\nabla f(x^1))$.

4. Данное движение продолжают до тех пор, пока не будет достигнуто окончание поиска, которое обычно определяют по величине нормы градиента $||\nabla f(x^k)||<\varepsilon$, где ε - заданная величина точности.

Сравнивая градиентный метод с методом покоординатного движения к оптимуму, можно заметить, что градиентный метод значительно более эффективен, поскольку на каждом шаге одновременно изменяется не одна, а сразу все координаты

$$x_j^{k+1}=x_j^k+\alpha\frac{\partial f(x_j^k)}{\partial x_j}\quad,\quad j=1,\ldots,n \tag{206}$$

Таким образом, движение к оптимуму осуществляется по кратчайшему направлению. Ниже приведен пример программы, в которой реализован метод простого градиентного поиска экстремума функции, использованной в предыдущих примерах.

```
import numpy as np
import matplotlib.pylab as plt
def f2(x1, x2, sigma):
    # Пример функции, которую мы хотим максимизировать
    err = np.random.normal(0, sigma)
    return -((x1 - 1) ** 2 + 1.5 * (x2 - 1) ** 2) + 1 + err  # Максимум в точке (1,1)
def gradient(x1, x2, sigma):
    # Вычисляем градиент функции
    err = np.random.normal(0, sigma)
    dfdx1 = -2 * (x1 - 1) + err  # Производная по x1
    dfdx2 = -3 * (x2 - 1) + err  # Производная по x2
    return np.array([dfdx1, dfdx2])  # Градиент
```

```
def gradient_move(starting_point, sigma, learning_rate=0.1, max_iterations=1000, tolerance=1e-3):
    x1, x2 = starting_point
    iters=[[x1,x2,f2(x1,x2,sigma)]]
    for i in range(max_iterations):
        grad = gradient(x1, x2, sigma)
        x1_new = x1 + learning_rate * grad[0]
        x2_new = x2 + learning_rate * grad[1]
        # Проверка на сходимость
        if np.linalg.norm([x1_new - x1, x2_new - x2]) < tolerance:
            break
        x1, x2 = x1_new, x2_new
        iters.append([x1,x2,f2(x1,x2,sigma)])
    return np.array(iters)
starting_point = (-1, -1)  # Начальная точка
sigma = 0.01  # Стандартное отклонение ошибки
iters = gradient_descent(starting_point, sigma)
x1_opt,x2_opt,y_opt=iters[-1]
print(f"Оптимальная точка: {x1_opt:.4f}, {x2_opt:.4f}")
print(f"Значение функции в оптимальной точке: {y_opt:.4f}")
print(f"Количество итераций: {len(iters)}")
x1_it=iters[:,0]
x2_it=iters[:,1]
x1 = np.linspace(-2, 3, 400)
x2 = np.linspace(-2, 3, 400)
X1, X2 = np.meshgrid(x1, x2)
# Вычисляем значения функции на сетке
Z = f2(X1, X2,0)
# Создаем контурный график и точки движения к экстремуму
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')
```

```
plt.colorbar(contour)
plt.title('Контурный график функции f(x)')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.scatter(x1_it, x2_it, color='green', marker='x', label='Точки итераций')
plt.scatter(x1_opt, x2_opt, color='blue', marker='o', label='Экстремум')
plt.legend()
plt.grid()
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.show()
```

В данной программе градиент заданной функции вычисляется аналитически по формулам (205) в функции gradient(x1, x2, sigma). Движение к точке оптимума осуществляется с помощью функции gradient_move, согласно вышеописанного алгоритма, с использованием формул (206).

Результаты вычислений:

```
Оптимальная точка: 0.9965, 1.0001
Значение функции в оптимальной точке: 1.0090
Количество итераций: 28
```

Точки движения к оптимума по градиенту показаны на графике (рис. 25). Как видно, по сравнению с методом Бокса-Уилсона, в данном случае движение к оптимуму осуществилось с большим количеством итераций, что связано с тем, что в данном случае величина шага на каждой итерации автоматически уменьшается по мере приближения к точке экстремума из-за с уменьшения градиента. Однако в результате в данном случае была достигнута большая точность вычислений.

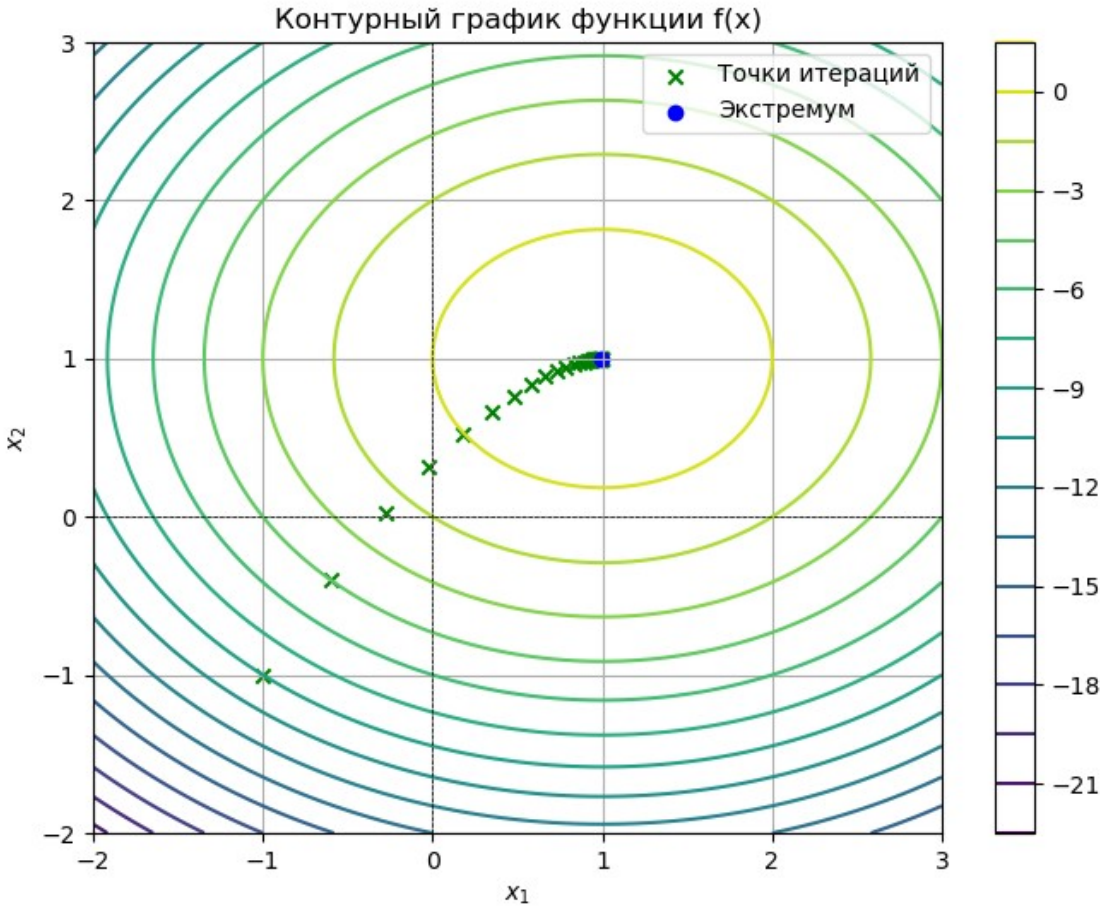

Рисунок 25. График, иллюстрирующий точки движения к оптимуму с использованием простого градиентного метода

Модификацией метода является движение с переменным шагом $x^{k+1}=x^k+\alpha_k\frac{\nabla f(x^k)}{\|\nabla f(x^j)\|}$, при котором величина шага α_k на каждой итерации определяется из решения задачи одномерной оптимизации в направлении вектора $\frac{\nabla f(x^k)}{\|\nabla f(x^j)\|}$ на текущей итерации.

Градиентные методы могут использоваться также и для решения задачи условной оптимизации. При наличии ограничений вида неравенств $\phi_i(x)\le 0$, $i=1,\ldots,m$ на изменение параметров целевой функции начальная точка выбирается так, чтобы она лежала в

пределах ограничений. После расчета следующей точки оценивается, не произошло ли нарушение ограничений; если нарушения нет, поиск продолжается. Когда какое-либо ограничение нарушено, производят перерасчет градиента с учетом ограничений. Одним из способов перерасчета градиента при наличии ограничений является метод обобщенного критерия. В соответствии с ним в качестве критерия оптимизации выбирается следующая функция:

$$Q(x)=f(x)-\alpha H(x) \quad , \tag{207}$$

где $H(x)=\sum_{i=1}^{m}\hat{\phi}_i(x)$, $\hat{\phi}_i(x)=\begin{cases}0 & при\ \phi_i\le 0\\ \phi_i & при\ \phi_i>0\end{cases}$, α - достаточно большая положительная константа, выбранная таким образом, чтобы условие

$\alpha\left|\frac{\partial H(x)}{\partial x_i}\right| >> \left|\frac{\partial f(x)}{\partial x_i}\right|$, $i=1,\ldots,k$ выполнялось всюду за пределами области ограничений.

В той области, где ограничения выполняются, $H(x)=0$ и обобщенный критерий совпадает с целевой функцией $f(x)$. Как только текущая точка поиска перейдет за какую-либо из границ ограничений, градиент обобщенного критерия уменьшается на достаточно большую величину $\alpha\nabla H(x)$:

$$\nabla Q(x)=\nabla f(x)-\alpha\nabla H(x) \quad . \tag{208}$$

В результате на следующем шаге направление движения изменится на нормальное к поверхности ограничений, и текущая точка снова попадет в допустимую область. В итоге движение к оптимуму будет происходить вдоль поверхности ограничений. Если безусловный экстремум целевой функции лежит за пределами области ограничений, то поиск закончится на поверхности ограничений или на линии ограничений (в двумерном случае) в точке, находящейся

на наименьшем расстоянии до точки безусловного экстремума. Схема движения к оптимуму при наличии ограничений изображена на рисунке 26.

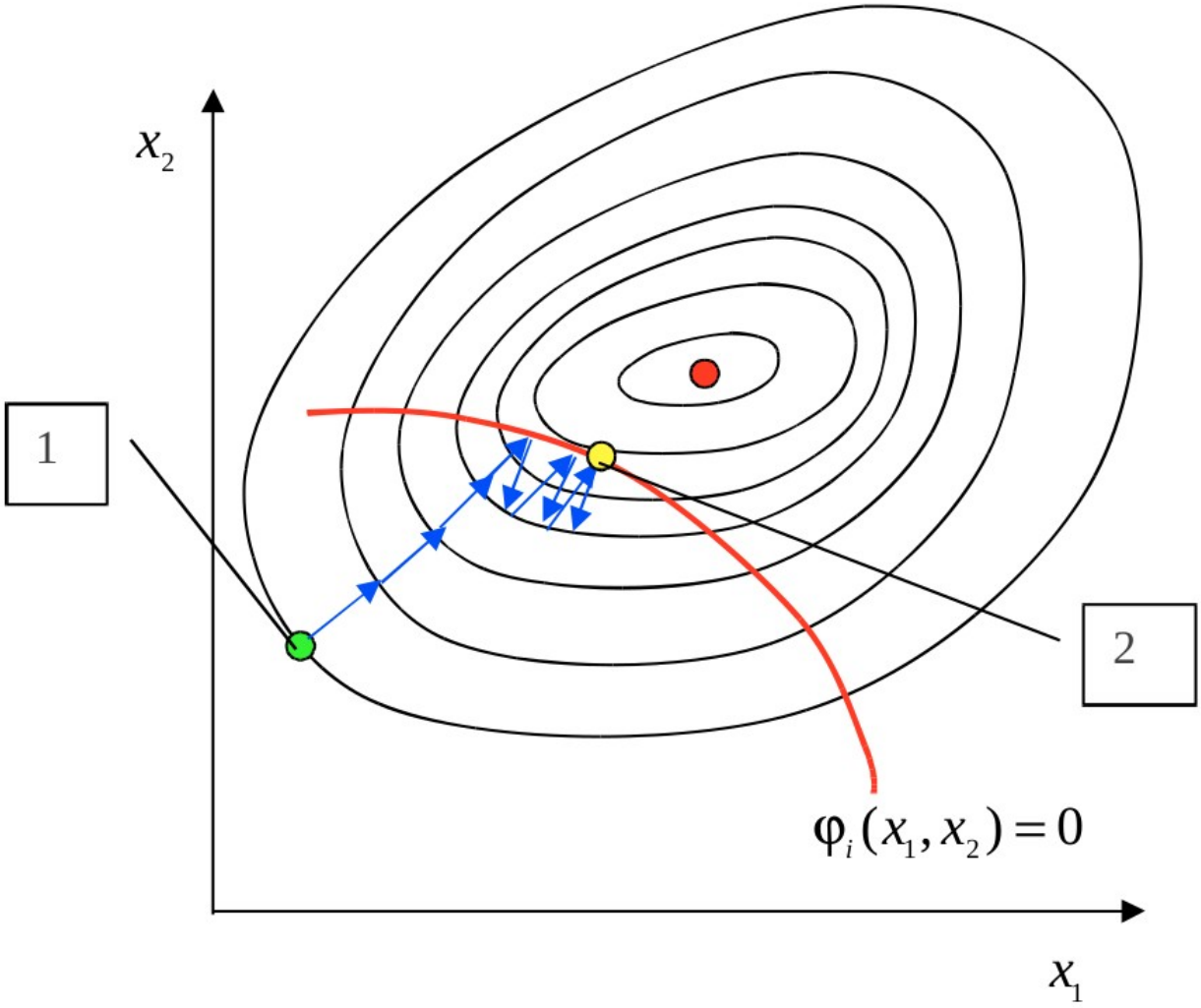

Рисунок 26. Схема движения к оптимуму при наличии ограничения

Точка 1 соответствует началу поиска, точка 2 соответствует окончанию поиска с достижением компромиссного экстремума. Из схемы видно, что в точке 2 процесс поиска экстремума фактически "зацикливается" и будет продолжаться бесконечно, поскольку направление вектора градиента целевой функции противоположно направлению антиградиента функции ограничений. Таким образом, после каждого шага в сторону возрастания градиента $f(x)$ текущая точка будет попадать за границу линии ограничений и по формуле (207) снова отбрасываться назад. В таких

случаях в алгоритме поиска необходимо предусматривать специальные средства для окончания поиска, так как условие заданной величины нормы градиента $||\nabla f(x^i)|| < \varepsilon$ работать не будет.

Существуют и другие варианты реализации численных методов условной оптимизации, в том числе и для симплексного метода.

Достоинством градиентных методов оптимизации является высокая скорость сходимости, что определяет их широкое практическое использование. Недостатками является низкая помехозащищенность по сравнению с методами нулевого порядка, а также необходимость в явном виде вычислять производные функции. В случае затруднений или невозможности явного вычисления градиента целевой функции, он может быть рассчитан численно с использованием разностных аппроксимаций производных.

Другая проблема для рассмотренных выше одношаговых градиентных методов заключается в плохой сходимости их для функций, поверхности которых имеют «овраги» (рис. 27) при поиске минимума или «хребты» при поиске максимума.

С целью устранения этого недостатка были созданы двухшаговые градиентные методы. Их преимущество заключается в том, что в процессе движения к оптимуму используется информация, получаемая не только на текущем шаге, но и на предшествующем. Среди различных двухшаговых методов наибольшую популярность приобрел метод сопряженных градиентов.

Свое название метод сопряженных градиентов получил вследствие того, что для случая квадратичной целевой функции $f(x) = (Ax, x) - (b, x)$, где A - положительно определенная матрица, последовательные направления движения $p^k = x^k - x^{k-1}$ удовлетворяют соотношению

$$(Ap^i, p^j) = 0 \quad , \quad i \neq j \quad .$$

Векторы, связанные данным соотношением, называются сопряженными или A-ортогональными (они ортогональны в метрике, задаваемой матрицей A). Доказано, что для квадратичной целевой функции метод сопряженных градиентов сходится за конечное число шагов.

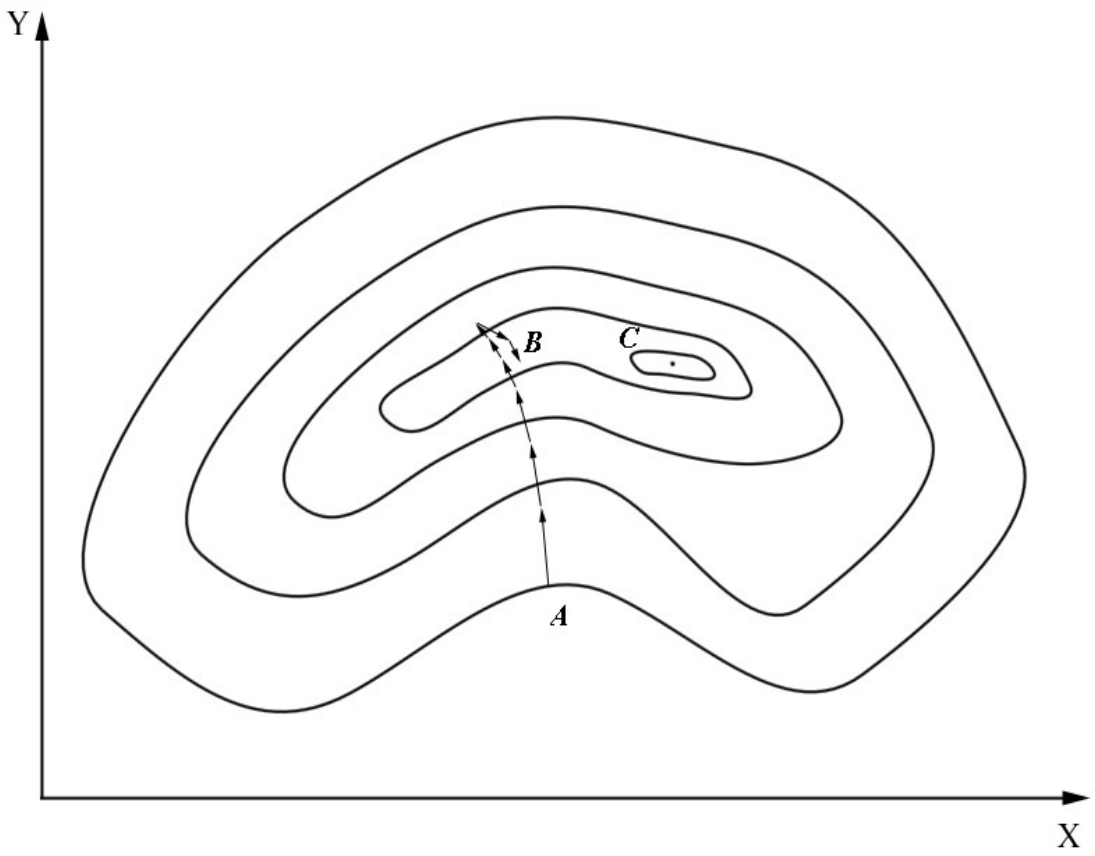

Рисунок 27. Схема движения простого градиентного метода для поверхности, имеющей "овраг" при поиске минимума или "хребет" при поиске максимума

Опишем наиболее простой вариант алгоритма метода сопряженных градиентов для поиска минимума целевой функции $f(x)$.

1. Инициализация:

- Выбрать начальную точку x_0 и вычислить начальный градиент $g_0 = \nabla f(x_0)$.

- Установить начальное направление $p_0 = -g_0$.

2. Итерационный процесс:

- Для каждой итерации k :

- Поиск размера шага. Найти оптимальный шаг α_k для минимизации функции вдоль направления p_k :

$$x_{k+1} = x_k + \alpha_k p_k$$

- Обновление градиента. Вычислит. новый градиент $g_{k+1} = \nabla f(x_{k+1})$.

- Обновление направления. Рассчитать новый вектор направления p_{k+1} с использованием предыдущего градиента и нового градиента. Обычно это делается по формуле:

$$beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$$

$$p_{k+1} = -g_{k+1} + \beta_k p_k$$

3. Условие остановки:

- Алгоритм продолжается до тех пор, пока норма градиента $\|g_k\|$ не станет меньше заданного порога, что указывает на достижение локального минимума.

Отличие описанного алгоритма от простого градиентного метода состоит также в том, что на каждом шаге движения к оптимуму решается задача одномерной оптимизации по поиску оптимальной величины шага t_k .

В качестве стандартной тестовой функции для алгоритмов оптимизации часто используется функция Розенброка, также известная как "функция банана"

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \quad (209)$$

где a и b обычно принимаются равными 1 и 100 соответственно. Оптимальное значение функции достигается в точке $(x, y) = (a, a^2)$.

Приведем пример программы, в которой решается задача поиска минимума функции Розенброка методом сопряженных градиентов и простым градиентным методом.

```
import numpy as np
import matplotlib.pylab as plt
# Функция Розенброка
def rosenbrock(x):
    a = 1
    b = 100
```

```
    return (a - x[0])**2 + b * (x[1] - x[0]**2)**2
# Градиент функции Розенброка
def rosenbrock_gradient(x):
    a = 1
    b = 100
    df_dx = -2 * (a - x[0]) - 4 * b * x[0] * (x[1] - x[0]**2)
    df_dy = 2 * b * (x[1] - x[0]**2)
    return np.array([df_dx, df_dy])
# Метод сопряженных градиентов
def conjugate_gradient(x0, tol=1e-6, max_iter=100):
    x = x0
    r = rosenbrock_gradient(x)
    p = -r
    k = 0
    iters_cg=[x.copy()]
    while np.linalg.norm(r) > tol and k < max_iter:
        alpha = line_search(x, p)
        x = x + alpha * p
        r_new = rosenbrock_gradient(x)
        beta = np.dot(r_new, r_new) / np.dot(r, r)
        p = -r_new + beta * p
        r = r_new
        k += 1
        iters_cg.append(x.copy())
    return (x,iters_cg)
# Линейный поиск для нахождения шага
def line_search(x, p):
    alpha = 1.0
    c = 1e-4
     while rosenbrock(x + alpha * p) > rosenbrock(x) + c * alpha * np.dot(rosenbrock_gradient(x),
p):
```

```
        alpha *= 0.5
    return alpha
# Простой метод градиентов
def gradient_descent(x0, tol=1e-6, max_iter=100, alpha=0.001):
    x = x0
    k = 0
    iters_gd=[x.copy()]
    while np.linalg.norm(rosenbrock_gradient(x)) > tol and k < max_iter:
        x = x - alpha * rosenbrock_gradient(x)
        k += 1
        iters_gd.append(x.copy())
    return (x,iters_gd)
# Начальная точка
x0 = np.array([-1.2, -1.0])
# Оптимизация с помощью метода сопряженных градиентов
rez_cg = conjugate_gradient(x0)
optimal_x_cg = rez_cg[0]
iters_cg=np.array(rez_cg[1])
optimal_value_cg = rosenbrock(optimal_x_cg)
# Оптимизация с помощью простого метода градиентов
rez_gd = gradient_descent(x0)
optimal_x_gd = rez_gd[0]
iters_gd=np.array(rez_gd[1])
optimal_value_gd = rosenbrock(optimal_x_gd)
# Результаты
print("Метод сопряженных градиентов:")
print("Оптимальная точка:", optimal_x_cg)
print("Значение функции в оптимальной точке:", optimal_value_cg)
print("\nПростой метод градиентов:")
print("Оптимальная точка:", optimal_x_gd)
print("Значение функции в оптимальной точке:", optimal_value_gd)
```

```
x1 = np.linspace(-2, 2, 400)
x2 = np.linspace(-2, 3, 400)
X1, X2 = np.meshgrid(x1, x2)
# Вычисляем значения функции на сетке
Z = rosenbrock([X1, X2])
# Создаем контурный график и точки движения к экстремуму
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=np.logspace(-1, 3, 8), cmap='viridis')
plt.title('Контурный график функции Розенброка')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.plot(iters_cg[:,0], iters_cg[:,1], color='red', marker='x', label='Метод сопряженных градиентов')
plt.plot(iters_gd[:,0], iters_gd[:,1], color='blue', marker='+', label='Градиентный метод')
plt.plot(x0[0], x0[1], color='green', marker='o', markersize=10, label='Стартовая точка')
plt.legend()
plt.grid()
plt.axhline(0, color='black', lw=0.5, ls='--')
plt.axvline(0, color='black', lw=0.5, ls='--')
plt.show()
```

В данной программе определяется функция Розенброка rosenbrock(x) по формуле (209) и ее градиент, который вычисляется аналитически в функции rosenbrock_gradient(x). Далее определяется функция conjugate_gradient, в которой реализован описанный выше алгоритм метода сопряженных градиентов и необходимая для нее функция line_search одномерного поиска по направлению вектора p. Затем определяется функция gradient_descent, в которой реализован метод простого градиентного поиска. Далее в программе решается задача минимизации функции Розенброка каждым методом и строятся контурные графики функции и точек итераций (рис 28). Результаты вычислений:

```
Метод сопряженных градиентов:
Оптимальная точка: (1.0014,1.0028)
Значение функции в оптимальной точке: 0.0000
```

Простой метод градиентов:

Оптимальная точка: (0.1624,0.0240)

Значение функции в оптимальной точке: 0.7022

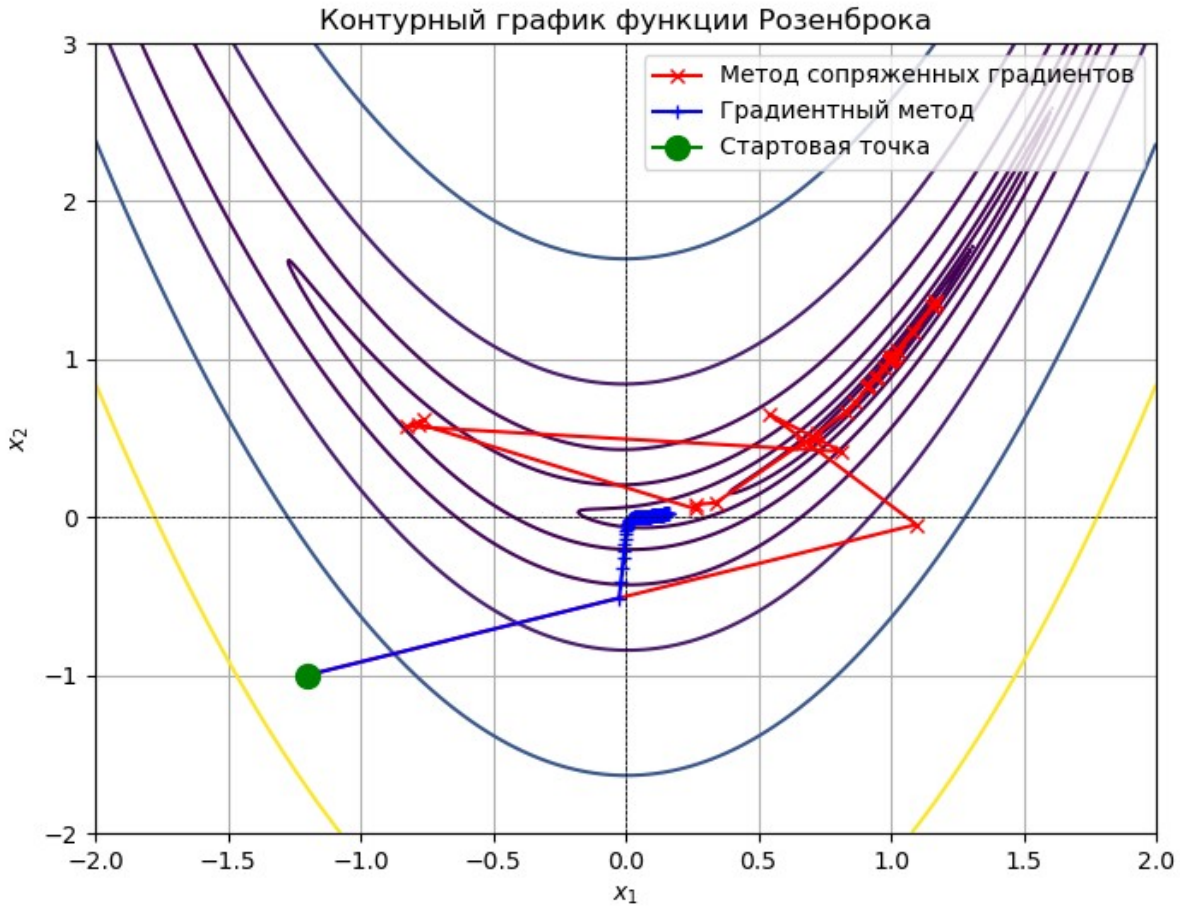

Рисунок 28. Контурный график функции Розенброка и точки итераций нахождения экстремума методом сопряженных градиентов и простым градиентным методом

В данном случае теоретический минимум функции Розенброка в точке с координатами (1,1) равен 0. Из результатов вывода программы видно, что метод сопряженных градиентов отлично справляется с задачей за 100 итераций, тогда как простой градиентный метод дает большую ошибку. Из графика видно, насколько сильно различаются траектории движения к оптимуму данных методов. Шаг первой итерации осуществляется по направлению простого градиента и совпадает у обоих методов. Однако далее траектория простого градиентного метода изменяется и

двигаясь по нормали к градиенту поиск «застревает» в овраге. В случае метода сопряженных градиентов траектория поиска имеет сложный характер. Сначала точки поиска как бы «отталкиваются» от «стенок» оврага, а затем траектория поиска проходит по дну оврага и заканчивается в точке оптимума функции с координатами, близкими к теоретическим.

В методах оптимизации второго порядка для определения направления поиска используются не первые, а вторые производные целевой функции. Это позволяет существенно увеличить скорость движения к оптимуму вблизи точки экстремума, когда первые производные малы и методы первого порядка становятся неэффективными. В методе Ньютона-Рафсона, относящемся к методам второго порядка, для определения координат целевой функции на каждом последующем шаге итерации используется полная матрица Гессе вторых частных производных целевой функции на предыдущем шаге

$$x^{k+1} = x^k + t_k H^{-1} \nabla f(x^k) \quad , \qquad (210)$$

где - матрица, обратная матрице Гессе на *k*-м шаге итерации, $H_{ij} = \frac{\partial^2 f(x^k)}{\partial x_i \partial x_j}$.

Вместе с тем методы второго порядка еще более чувствительны к ошибкам в вычисления функции, чем методы первого порядка. Поэтому в случаях отсутствия возможности аналитически вычислить вторые производные целевой функции их применять не рекомендуется.

В библиотеке scipy.optimiz.minimize реализован целый набор функций на основе разных алгоритмов, позволяющих численно решать задачу безусловной и условной оптимизации, нелинейного регрессионного анализа и решения нелинейных уравнений и их систем. В качестве примера ниже приведен текст простой программы, в которой производится сравнение нескольких методов оптимизации тестовой функции Розенброка.

```
import numpy as np
from scipy.optimize import minimize
# Функция Розенброка
def rosenbrock(x):
    a = 1
    b = 100
    return (a - x[0])**2 + b * (x[1] - x[0]**2)**2
```

```
# Определим начальную точку
x0 = np.array([-1, -1])
# Создадим список методов оптимизации
methods = ['Nelder-Mead', 'Powell', 'COBYLA', 'CG', 'BFGS', 'L-BFGS-B' ]
#Решим задачу оптимизации
for method in methods:
    result = minimize(rosenbrock, x0, method=method)
        print(f"Метод: {method}, Результат: {result.success}, Сообщения: {result.message}, Количество вычислений функции: {result.nfev}\n")
    print(result,'\n')
```

В данной программе производится сравнение шести методов оптимизации, из которых первые три ('Nelder-Mead', 'Powell', 'COBYLA') относятся к методам нулевого порядка, метод сопряженных градиентов ('CG') - двухшаговый метод первого порядка и два последних метода - методы второго порядка. Результаты работы программы:

```
Метод: Nelder-Mead, Результат: True, Сообщения: Optimization terminated successfully., Количество вычислений функции: 125
      message: Optimization terminated successfully.
      success: True
       status: 0
          fun: 5.309343918637161e-10
            x: [ 1.000e+00  1.000e+00]
          nit: 67
         nfev: 125
 final_simplex: (array([[ 1.000e+00,  1.000e+00],
                        [ 1.000e+00,  9.999e-01],
                        [ 1.000e+00,  1.000e+00]]), array([ 5.309e-10,  1.412e-09,  5.059e-09]))

Метод: Powell, Результат: True, Сообщения: Optimization terminated successfully., Количество вычислений функции: 47
 message: Optimization terminated successfully.
 success: True
  status: 0
```

fun: 0.0
x: [1.000e+00 1.000e+00]
nit: 2
direc: [[0.000e+00 1.000e+00]
[1.000e+00 1.000e+00]]
nfev: 47

Метод: COBYLA, Результат: False, Сообщения: Maximum number of function evaluations has been exceeded., Количество вычислений функции: 1000

message: Maximum number of function evaluations has been exceeded.
success: False
status: 2
fun: 0.02800798626432505
x: [8.328e-01 6.928e-01]
nfev: 1000
maxcv: 0.0

Метод: CG, Результат: False, Сообщения: Desired error not necessarily achieved due to precision loss., Количество вычислений функции: 210

message: Desired error not necessarily achieved due to precision loss.
success: False
status: 2
fun: 7.45592153442599e-12
x: [1.000e+00 1.000e+00]
nit: 21
jac: [4.241e-05 -1.925e-05]
nfev: 210
njev: 66

Метод: BFGS, Результат: True, Сообщения: Optimization terminated successfully., Количество вычислений функции: 120

message: Optimization terminated successfully.

```
 success: True
  status: 0
     fun: 1.9949932849175868e-11
       x: [ 1.000e+00  1.000e+00]
     nit: 31
     jac: [ 2.786e-07 -1.273e-07]
hess_inv: [[ 5.084e-01  1.016e+00]
           [ 1.016e+00  2.037e+00]]
    nfev: 120
    njev: 40
```

Метод: L-BFGS-B, Результат: True, Сообщения: CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL, Количество вычислений функции: 99

```
 message: CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
 success: True
  status: 0
     fun: 9.141040468035814e-12
       x: [ 1.000e+00  1.000e+00]
     nit: 26
     jac: [ 5.219e-07 -2.787e-07]
    nfev: 99
    njev: 33
hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

Как можно видеть, полностью не справился с задачей оптимизации тестовой функции Розенброка метод COBYLA. В случае метода сопряженных градиентов программа выдала сообщение о неуспешном завершении процедуры поиска, но, тем не менее, координаты экстремума были выданы верно.

В качестве примера решения задачи условной оптимизации с использованием функции библиотеки scipy.optimiz.minimize ниже приведен пример программы оптимизации целевой функции в виде уравнения параболической регрессии (189), график поверхности которой приведен на рисунке 16, с ограничениями вида неравенств.

```
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
# Определяем функцию параболической регрессии
def func(x, b):
    return (b[0] + b[1]*x[0] + b[2]*x[1]+ b[3]*x[0]**2 + b[4]*x[1]**2 + b[5]*x[0]*x[1])
# Определяем ограничения
def constraint1(x):
    return x[0]**2 +x[1] - 0.5 #>=0
def constraint2(x):
    return x[0]+x[1] -0.25 #>=0
# Начальное приближение
x0 = [0, 0]
# Коэффициенты
b0=1;b1=1;b2=1;b11=3;b22=3;b12=1
b_i = [b0, b1, b2, b11, b22, b12]
#Решение задачи безусловной оптимизации
res_0 = minimize(func, x0, args=(b_i))
extr_0=[res_0.x[0],res_0.x[1]]
print(f"Координаты безусловного экстремума: {extr_0[0]:.4f}, {extr_0[1]:.4f}\n Значение функции в точке экстремума {func(res_0.x, b_i):.4f}")
# Минимизация с ограничениями
res_1 = minimize(func, x0, args=(b_i), constraints=({'type': 'ineq', 'fun': constraint1}, {'type': 'ineq', 'fun': constraint2}))
extr_1=[res_1.x[0],res_1.x[1]]
print(f"\nКоординаты условного экстремума: {extr_1[0]:.4f}, {extr_1[1]:.4f}\n Значение функции в точке условного экстремума {func(res_1.x, b_i):.4f}")
#Построение графиков
x0 = np.linspace(-1.5, 1.5, 100)
x1 = np.linspace(-1.5, 1.5, 100)
X0, X1 = np.meshgrid(x0, x1)
Y = func([X0, X1], b_i)
```

```
plt.figure(figsize=(8, 6))
def fextr(x,extr):
    return (x[0]-extr[0])**2 + (x[1]-extr[1])**2
Y1=constraint1([X0, X1])
Y2=constraint2([X0, X1])
Yextr=fextr([X0, X1],extr_1)
# Строим контурный график
plt.contour(X0, X1, Y, levels=[0.9,1,2,3], colors='b')
plt.contour(X0, X1, Y1, levels=[0], colors='r')
plt.contour(X0, X1, Y2, levels=[0], colors='g')
plt.contour(X0, X1, Yextr, levels=[0.005], colors='k')
plt.plot(extr_0[0], extr_0[1], color='green', marker='o', markersize=10, label='Безусловный экстремум')
plt.plot(extr_1[0], extr_1[1], color='red', marker='o', markersize=10, label='Условный экстремум')
plt.title('Условная оптимизация функции параболической регрессии')
plt.xlabel('x0')
plt.ylabel('x1')
plt.legend()
plt.show()
```

В данной программе определяются два ограничения вида неравенств:

1. Нелинейное ограничение constraint1(x) вида $x_0^2 + x_1 - 0.5 \geq 0$

2. Линейное ограничение constraint2(x) вида $x_0 + x_1 - 0.25 \geq 0$.

Решение задачи безусловной оптимизации осуществляется с использованием функции

minimize(func, x0, args=(b_i)).

Задача условной оптимизации решается с использованием той же функции, но в число параметров добавлены ссылки на ограничения:

minimize(func, x0, args=(b_i), constraints=({'type': 'ineq', 'fun': constraint1}, {'type': 'ineq', 'fun': constraint2})).

На печать выводятся найденные координаты безусловного и условного экстремумов:

Координаты безусловного экстремума: -0.1429, -0.1429

Значение функции в точке экстремума 0.8571

Координаты условного экстремума: -0.2071, 0.4571

Значение функции в точке условного экстремума 1.9108

На графике (рис. 29) видно, что точка условного экстремума находится на пересечении линий ограничений.

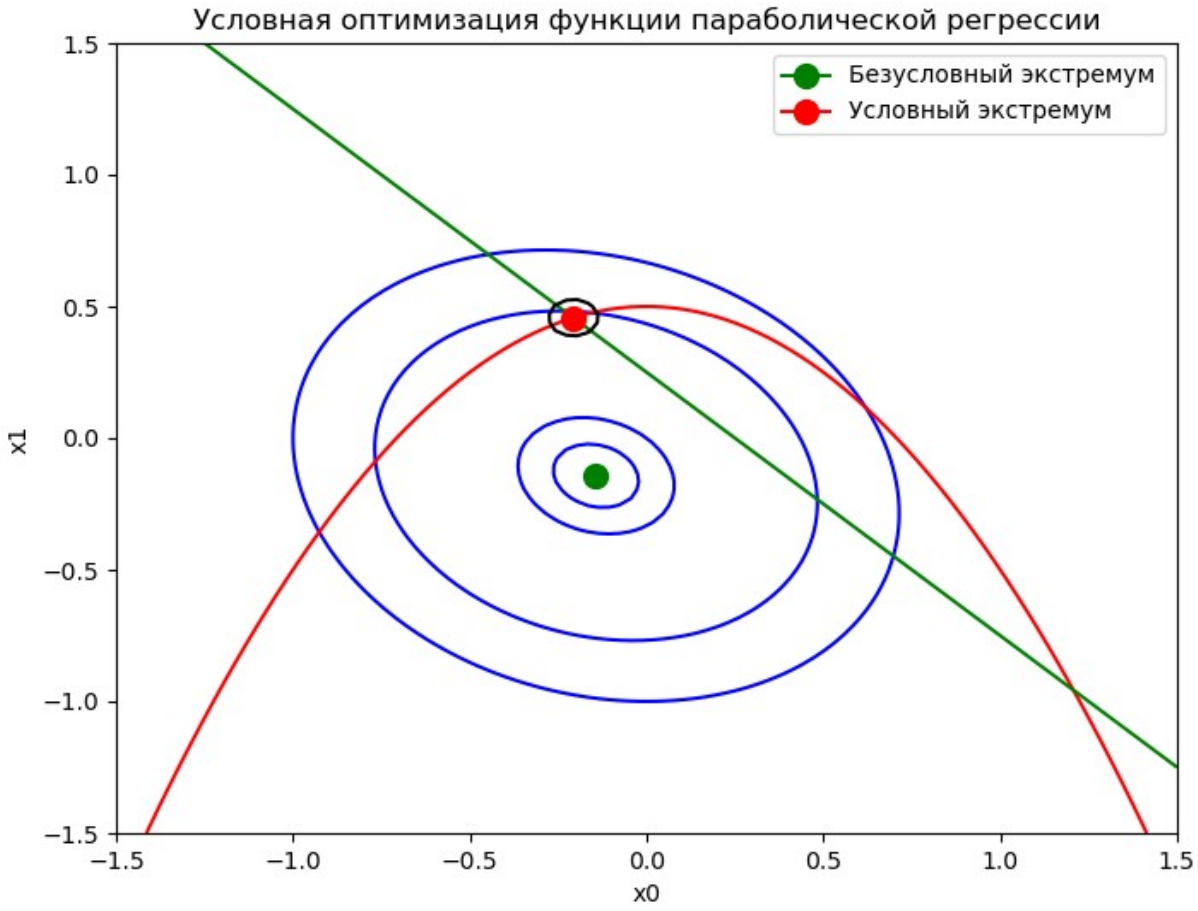

Рисунок 29. Результаты решения задачи условной оптимизации для функции параболической регрессии

4.3.5. Методы глобальной оптимизации

Рассмотренные методы пошагового движения к экстремуму не пригодны, если целевая функция имеет несколько локальных экстремумов. В этом случае любой из вышеперечисленных методов будет сходиться к ближайшему локальному экстремуму, в зависимости от координат начальной точки поиска. Для решения задачи отыскания глобального экстремума функции необходимо использованием специальных методов. Одним из простейших таких методов является метод случайного поиска, называемый также методом Монте-Карло. Практически его реализация возможна лишь с применением достаточно быстродействующих ЭВМ, позволяющих за короткое время вычислить достаточно большое количество значений целевой функции (10^4 и более). Эффективность его проявляется для задач большой размерности и для тех функций, для которых обычные методы, такие как градиентный, не дают результатов. Примером таких функций являются функции, имеющие несколько локальных экстремумов и экстремумы типа "оврагов", в которых градиентный поиск легко "зацикливается". Метод Монте-Карло естественным образом может применяться как для задач безусловной оптимизации, так и для задач условной оптимизации. Для ускорения сходимости можно строить также комбинированные алгоритмы, сочетая метод Монте-Карло с другими методами оптимизации.

Рассмотрим наиболее простой вариант алгоритма случайного поиска.

1. Генерируется случайный вектор - точка x^0, координаты которой $(x_1^0,\ldots,x_k^0)$ представляют собой случайные числа, равномерно распределенные в интервалах области допустимых значений независимых переменных. В точке x^0 вычисляется значение целевой функции $f(x^0)$.

2. Генерируется другой случайный вектор - точка x^1, в которой вычисляется значение целевой функции $f(x^1)$.

3. Проверяется неравенство $f(x^1) > f(x^0)$. Если неравенство выполняется (новая точка оказалась удачной), то точку x^1 принимают за новое значение x^0 и поиск продолжают

повторением п. 2. Если неравенство не выполняется (новая точка хуже предыдущей), точку x^1 не сохраняют и продолжают поиск повторением п. 2.

После достаточного числа итераций (обычно не менее 10^4) поиск сходится к абсолютному максимуму функции $f(x)$. В случае поиска минимуму целевой функции на шаге 3 проверяется неравенство $f(x^1) < f(x^0)$. Если необходимо решать задачу условной оптимизации, то на каждом шаге также осуществляется проверка выполнения условий ограничений.

В библиотеке scipy.optimize имеются методы глобальной оптимизации, более эффективные по сравнению с простым методом Монте-Карло. В качестве стандартной тестовой функции для методов глобальной оптимизации часто используется функция Акли. Она представляет собой многомодальную функцию с множеством локальных минимумов и одним глобальным минимумом, что делает её сложной задачей для алгоритмов оптимизации. Функция имеет следующий вид

$$f(x_0, x_1) = -20 \cdot \exp\left(-0.2 \cdot \sqrt{0.5 \cdot (x_0^2 + x_1^2)}\right) - \exp\left(0.5 \cdot (\cos(2\pi x_0) + \cos(2\pi x_1))\right) + e + 20 \qquad (211)$$

В библиотеке scipy.optimize имеется несколько функций глобальной оптимизации, начиная от самых простых (метод сканирования по области определения) и включая современные, основанные на эффективных алгоритмах глобального поиска. Ниже приводится программа оптимизации функции Акли описанным выше алгоритмом Монте-Карло, а также с использованием нескольких эффективных функций библиотеки scipy.optimize.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import differential_evolution,shgo,dual_annealing,direct
# Определение функции Акли
def ackley(x):
    a = 20
    b = 0.2
```

```
    c = 2 * np.pi
    sum1 = x[0]**2 + x[1]**2
    sum2 = np.cos(c * x[0]) + np.cos(c * x[1])
    return -a * np.exp(-b * np.sqrt(sum1 / 2)) - np.exp(sum2 / 2) + a + np.exp(1)
# Генерация случайных точек для метода Монте-Карло
def monte_carlo_optimization(func, bounds, n_iter=10000):
    best_x = None
    best_y = float('inf')
    for _ in range(n_iter):
        x = np.random.uniform(bounds[0], bounds[1], 2)
        y = func(x)
        if y < best_y:
            best_y = y
            best_x = x
    return best_x, best_y
# Оптимизация с использованием scipy.optimize
def scipy_optimization(method):
    result = method
    return result.x, result.fun
# Задаем границы
bounds = [-5, 5]
glob_bounds=((-5, 5),(-5, 5))
pars=(ackley,glob_bounds)#Параметры функций scipy
#Создаем словарь методов глобальной оптимизации scipy
scipy_methods={"differential_evolution":differential_evolution(*pars),
       "direct": direct(*pars), "shgo":shgo(*pars), "dual_annealing":dual_annealing(*pars)}
# Поиск оптимума методом Монте-Карло
best_x_mc, best_y_mc = monte_carlo_optimization(ackley, bounds)
print(f"Метод Монте-Карло,координаты: {best_x_mc}, значение функции: {best_y_mc:.4f}")
# Поиск оптимума с использованием scipy
for method in scipy_methods.keys():
```

```
    best_x_scipy, best_y_scipy = scipy_optimization(scipy_methods[method])
    print(f"Метод scipy: {method},координаты: {best_x_scipy}, значение функции: {best_y_scipy:.4f}")
# Построение 3D графика функции
x = np.linspace(bounds[0], bounds[1], 100)
y = np.linspace(bounds[0], bounds[1], 100)
X, Y = np.meshgrid(x, y)
Z = ackley([X, Y])
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(121, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.7)
ax.scatter(best_x_mc[0], best_x_mc[1], best_y_mc, color='r', s=100, label='MC Оптимум')
ax.scatter(best_x_scipy[0], best_x_scipy[1], best_y_scipy, color='b', s=100, label='SciPy Оптимум')
ax.set_title('3D график функции Акли')
ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('f(x)')
ax.legend()
# Построение контурного графика
ax2 = fig.add_subplot(122)
ax2.contour(X, Y, Z, levels=50, cmap='viridis')
ax2.scatter(best_x_mc[0], best_x_mc[1], color='r', s=100, label='MC Optimum')
ax2.scatter(best_x_scipy[0], best_x_scipy[1], color='b', s=100, label='SciPy Optimum')
ax2.set_title('Контурный график функции Акли')
ax2.set_xlabel('X1')
ax2.set_ylabel('X2')
ax2.legend()
plt.show()
```

В данной программе определена функция Акли ackley(x) и функция , реализующая простой алгоритм случайного поиска monte_carlo_optimization. Для оптимизации методами библиотеки scipy.optimize создана функция-обертка scipy_optimization(method), в которую в качестве параметра method. передается ссылка на конкретную функцию оптимизации из словаря scipy_methods, содержащего названия функций и команды для их вызова. Далее запускается алгоритм Монте-Карло и организуется цикл по всем ключам словаря scipy_methods, запускающий каждую функцию и печатающий ее результат. В заключительной части выводятся трехмерный и контурный графики функции Акли с отмеченными точками координат экстремума, найденных методом Монте-Карло и с помощью функции из библиотеки scipy.optimize. Графики приведены на рисунке 30.

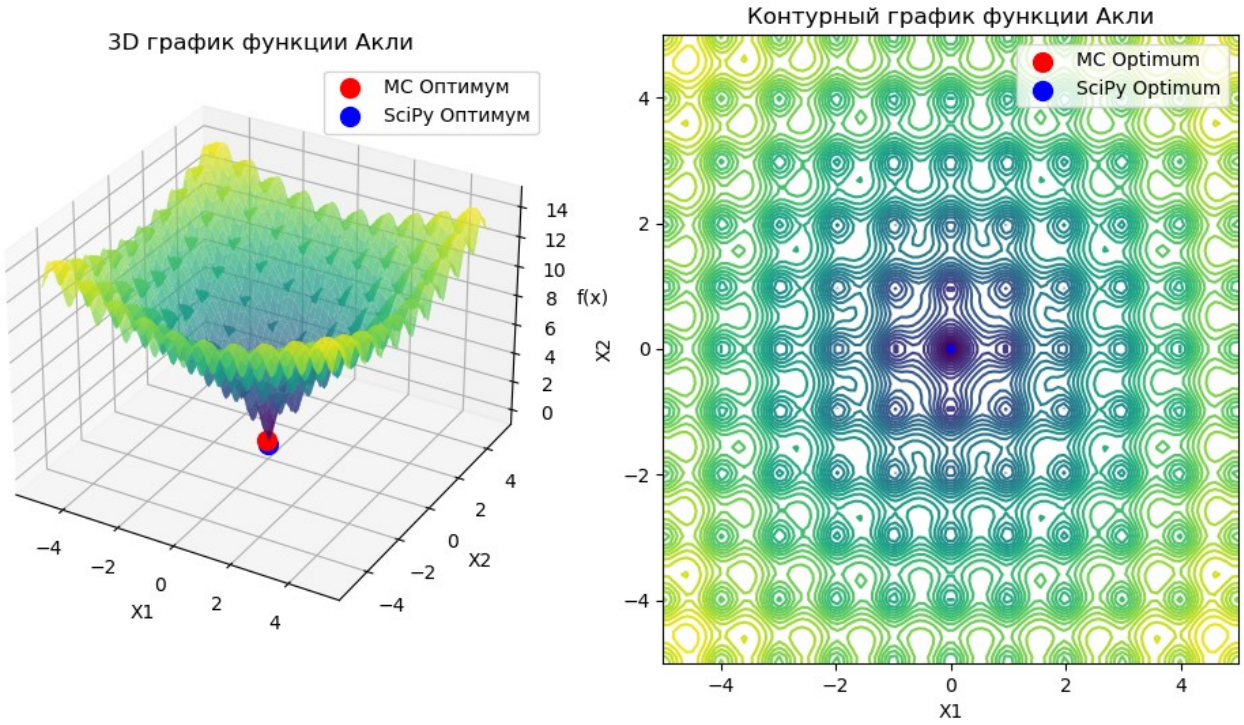

Рисунок 30. Трехмерный и контурный графики функции Акли и точки найденных экстремумов

Ниже представлен вывод результатов работы программы.

Метод Монте-Карло,координаты: [-0.05979576 -0.02090083], значение функции: 0.2828
Метод scipy: differential_evolution, координаты: [0. 0.], значение функции: 0.0000
Метод scipy: direct, координаты: [0. 0.], значение функции: 0.0000

Метод scipy: shgo, координаты: [-9.02894984e-11 -9.02900391e-11], значение функции: 0.0000

Метод scipy: dual_annealing, координаты: [-3.42499351e-09 2.19175699e-09], значение функции: 0.0000

Функция Акли имеет глобальный минимум в точке с координатами (0,0), равный 0. Как видно из вывода результатов, все протестированные алгоритмы библиотеки scipy.optimize решают с высокой точностью задачу глобальной оптимизации, тогда как при числе итераций, равном 10000, метод Монте-Карло дает некоторую ошибку.

Глава 5 Нелинейная регрессия и нейронные сети

Нелинейный регрессионный анализ является важным статистическим методом для моделирования сложных нелинейных зависимостей между переменными. В отличие от линейной регрессии, нелинейная регрессия позволяет использовать более гибкие функциональные формы для описания взаимосвязей [37]. Нелинейность может относиться как к самим переменным, так и к параметрам (например, показательные, степенные, логарифмические, тригонометрические, гауссовы функции и др.). В ряде случаев для упрощения анализа применяют линеаризацию — преобразование данных такими способами, чтобы задача нелинейной регрессии стала задачей линейной регрессии. Например, для показательной зависимости $y = ae^{bx}$ после логарифмирования задача становится линейной по отношению к параметрам. Однако, такой подход требует особой осторожности, так как может исказить структуру ошибок в модели и повлиять на свойства оценок. Нелинейный регрессионный анализ находит широкое применение в различных областях, включая биологию, физику, экономику и инженерные науки [38, 39]. Он позволяет моделировать такие явления, как насыщение, пороговые эффекты и другие нелинейные процессы, которые невозможно адекватно описать с помощью линейных моделей. В последние годы активно развиваются методы непараметрической нелинейной регрессии, основанные на машинном обучении, такие как нейронные сети и методы ансамблевого обучения [40, 41]. Эти подходы позволяют моделировать еще более сложные нелинейные зависимости без необходимости явного задания функциональной формы. Искусственные нейронные сети, моделирующие работу биологических нейронов, открыли новые горизонты в решении задач нелинейного моделирования и прогнозирования. Благодаря своей универсальности, способность к адаптации и обучению по прецеденту, нейронные сети успешно применяются в задачах регрессии, анализа временных рядов, обработки изображений и других областях, где традиционные методы оказываются малоэффективными или неприменимыми.

5.1 Уравнение нелинейной регрессии

В разделе 3.3 были рассмотрены уравнения регрессии, линейные по параметрам. Отклики y_j линейной по параметрам модели можно представить в виде

$$y_j = \theta^T f(x_j) + \varepsilon_j \qquad (212)$$

где ε_j — случайные величины, распределение которых предполагается нормальным с нулевым математическим ожиданием $E\,\varepsilon_j = 0$ и диагональной ковариационной матрицей $E\,\varepsilon_j \varepsilon_k = \sigma^2 \delta_{jk}$, $\theta = (\theta_1, \ldots, \theta_m)^T$ — вектор неизвестных параметров из $\mathbb{R}^m$, $f(x) = (f_1(x), \ldots, f_m(x))^T$ - вектор заданных, линейно независимых на множестве X функций.

В матричных обозначениях $Y = (y_1, \ldots, y_n)^T$, $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)^T$, $F = (f_1(x_j), \ldots, f_m(x_j))_{j=1}^n$ система (212) записывается в виде

$$Y = F\,\theta + \varepsilon \quad , \tag{213}$$

где $EY = F\,\theta$ и ковариационная матрица DY равна $\sigma^2 I_n$, I_n — единичная матрица.

Оценки $\hat{\theta}$ неизвестных параметров θ по методу наименьших квадратов вычисляются согласно

$$\hat{\theta} = \underset{\theta \in \Theta}{argmin} \sum_{j=1}^{n} \sigma_j^{-2} \left| y_j - \eta(x_j, \theta) \right|^2 \tag{214}$$

или в матричных обозначениях

$$\hat{\theta} = \underset{\theta \in \Theta}{Argmin} (Y - F\,\Theta)^T (Y - F\,\Theta) \tag{215}$$

Решение задачи (215) сводится к известной формуле (159) регрессионного анализа, которая в обозначениях (213) имеет вид

$$\hat{\theta} = (F^T F)^{-1} F^T Y \tag{216}$$

Дисперсия адекватности модели s^2, являющаяся несмещенной оценкой дисперсии σ^2 при этом вычисляется по формуле $s^2 = SS_{reg}/(n-m)$, где

$$SS_{reg} = (Y - F\,\hat{\theta})^T (Y - F\,\hat{\theta}) \quad . \tag{217}$$

Для модели регрессии, нелинейной по параметрам, вместо представления откликов в виде (213) используется представление

$$Y = \mathrm{H}(X, \theta) + \varepsilon \quad , \tag{218}$$

где $\mathrm{H}(X, \theta) = (\eta(x_1, \theta), \ldots, \eta(x_n, \theta))^T$ - вектор значений заданной нелинейной функции $\eta(x, \theta)$ в точках x_j с параметрами θ. Такая модель называется также уравнением нелинейной регрессии.

Формулировка (215) метода наименьших квадратов в данном случае принимает вид

$$\hat{\theta}=\underset{\theta\in\Theta}{Argmin}\left(Y-\mathrm{H}\left(X,\Theta\right)\right)^{T}\left(Y-\mathrm{H}\left(X,\Theta\right)\right) \tag{219}$$

Система нормальных уравнений МНК при этом становится нелинейной, поэтому использовать простую формулу (216) для вычисления оценок параметров не удается и для решения данной задачи необходимо применять численные методы оптимизации, рассмотренные в предыдущей главе. Формула (217) для нелинейной модели имеет вид

$$SS_{reg}=\left(Y-\mathrm{H}\left(X,\hat{\theta}\right)\right)^{T}\left(Y-\mathrm{H}\left(X,\hat{\theta}\right)\right) \quad . \tag{220}$$

Таким образом, постановка задачи вычисления оценок коэффициентов нелинейной регрессии (219) аналогична постановке данной задачи для линейной регрессии (215) и разница заключается лишь в способе ее решения. Для решения данной задачи в случае нелинейной регрессии в библиотеке scipy.optimize имеются функции least_squares и curve_fit, в которых реализованы численные методы оптимизации.

В качестве примера рассмотрим задачу оценки параметров нелинейного уравнения регрессии вида

$$y_j=\theta_0\exp\left(\theta_1 x_j\right)+\theta_2+\varepsilon_j \quad , \tag{221}$$

где $\theta=\left(\theta_0,\theta_1,\theta_2\right)$ - параметры уравнения, где ε_j — случайные величины, в соответствии с обозначениями к (212). Подобные уравнения часто встречаются при аппроксимации свойств материалов, а также в химической кинетике.

Ниже приведен пример программы, в которой вычисляются оценки $\hat{\theta}$ коэффициентов регрессии (221) с использованием функции curve_fit.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
def func(x, *theta):
   return theta[0]*np.exp(theta[1]*x)+theta[2]
sigma=1.5
N=50
xdata = np.linspace(0, 3, N)
```

```
theta=np.array([2.,1.,0.5])
theta0=theta*10
ydata=func(xdata, *theta)+np.random.normal(0, sigma, N)
plt.plot(xdata, ydata, 'b-', label='Данные')
theta_opt, cov_matrix = curve_fit(func, xdata, ydata,p0=theta0)
print(f'Вычисленные значения параметров \n {theta_opt}')
print(f'Ковариационная матрица: \n {cov_matrix}')
plt.plot(xdata, func(xdata, *theta_opt), 'r-',label='Аппроксимация')
plt.xlabel('x')
plt.ylabel('Данные и регрессия')
plt.legend()
plt.show()
```

В данной программе определена функция регрессии func(x, *theta), в которую передается значение аргумента x и параметры θ. Символ «*» перед обозначением параметров обеспечивает распаковку массива при его передаче в качестве параметра функции. Далее в программе создаются массивы значений аргумента xdata и смоделированные в этих точках отклики ydata с нормально распределенной ошибкой с нулевым математическим ожиданием и стандартным отклонением sigma. Истинные значения параметров theta задаются как np.array([2.,1.,0.5]), а начальные значения для поиска theta0 задаются путем умножения истинных на достаточно большое число (в данном примере 10). В функцию curve_fit(func, xdata, ydata,p0=theta0) передаются в качестве параметров ссылка на имя функции регрессии func, массивы экспериментальных точек xdata, ydata и начальное значение параметров p0=theta0. В результате поиска функция возвращает найденные значения параметров theta_opt (в виде array) и ковариационную матрицу cov_matrix (2-D array), которые выводятся затем на печать. В заключительной части программы строятся графики исходных данных и функции регрессии. Ниже приводится пример вывода результатов работы программы и график (рис.).

```
Вычисленные значения параметров:
 [2.16385096 0.9784527  0.04928109]
Ковариационная матрица:
 [[ 0.16396125 -0.02438773 -0.30437862]
 [-0.02438773  0.00366866  0.0440834 ]
```

```
[-0.30437862  0.0440834   0.64507027]]
```

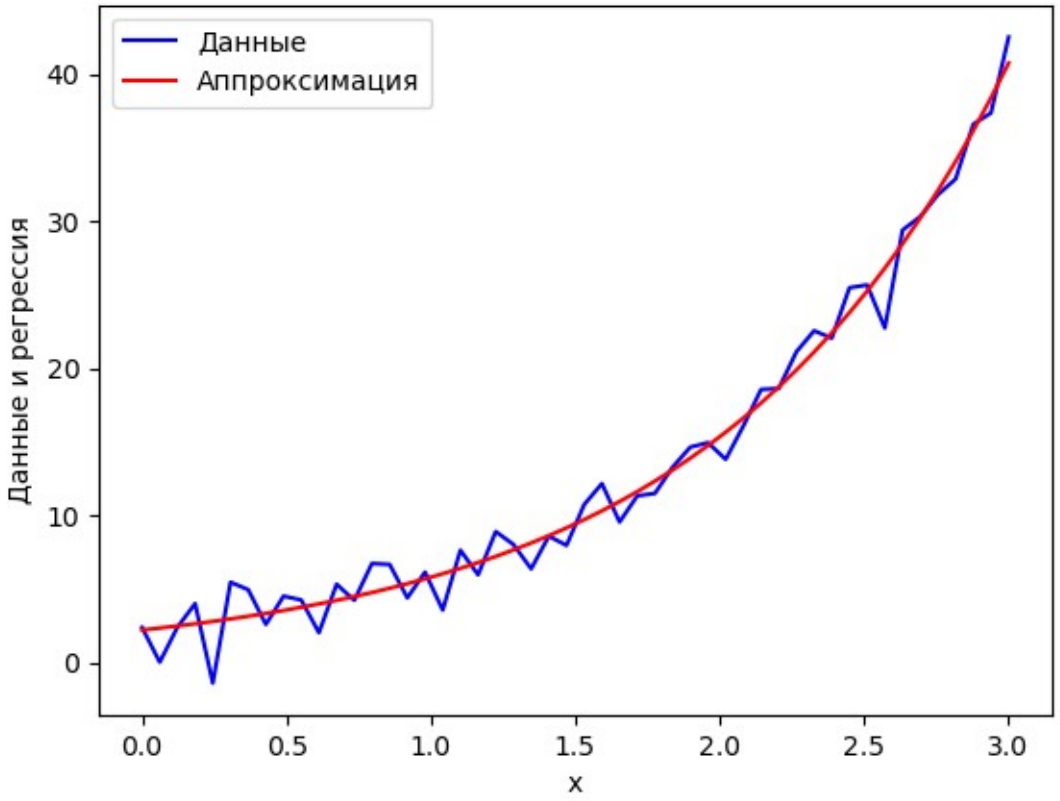

Рисунок 31. График исходных данных (ломаная) и аппроксимирующая их кривая регрессии по уравнению (221).

5.2. Логистическая регрессия

Логистическая регрессия — это статистическая модель, используемая для прогнозирования вероятности возникновения некоторого события. Она особенно полезна в задачах бинарной классификации, где зависимая переменная может принимать только два значения, например, «да» или «нет», «0» или «1».

1. Математическая основа: Логистическая регрессия основана на логистической функции, которая преобразует любое вещественное число в значение между 0 и 1. Это значение интерпретируется как вероятность того, что событие произойдет. Логистическая функция (или

сигмоидальная функция) — это одна из важнейших и наиболее часто применяемых функций активации в искусственных нейронных сетях. Она относится к классу S-образных (sigmoid) функций и используется для преобразования взвешенной суммы входов нейрона в ограниченный диапазон значений от 0 до 1, что удобно для моделирования вероятностей и нормализации выходных сигналов.

Математическое определение логистической функции:

$$f(x)=\frac{1}{1+e^{-x}} \tag{222}$$

где x — аргумент функции (например, взвешенная сумма входов нейрона).

Свойства логистической функции:

- Является гладкой, монотонно возрастающей и нелинейной.

- Ограничена сверху и снизу: при $x \to -\infty$ значение функции стремится к 0, а при $x \to +\infty$ — к 1.

- Применяется для нормализации выходов нейронов, часто используется на выходном слое для реализации вероятностной интерпретации результата.

- Ее производная легко вычисляется и выражается через саму функцию:

$$f'(x)=f(x)\cdot(1-f(x)) \ . \tag{223}$$

2. Независимые переменные: Модель использует множество независимых переменных (признаков) для прогнозирования вероятности события. Эти переменные могут быть числовыми или категориальными.

3. Оптимизация: В отличие от линейной регрессии, которая использует метод наименьших квадратов для оптимизации, логистическая регрессия оптимизируется методом максимального правдоподобия, часто с помощью функции бинарной кросс-энтропии.

4. Применение: Логистическая регрессия широко используется в различных областях, таких как медицина, финансы и социология, для оценки вероятности возникновения определенных событий на основе набора независимых переменных.

Примеры использования

- Кредитный скоринг: Оценка вероятности одобрения кредита на основе таких факторов, как возраст, доход и профессия заемщика.

- Медицинская диагностика: Прогнозирование вероятности заболевания на основе медицинских показателей.

Различия с линейной регрессией

- Выходные данные: Линейная регрессия предсказывает непрерывные значения, тогда как логистическая регрессия дает вероятность события.

- Математический подход: Линейная регрессия использует линейную модель, а логистическая — сигмоидальную кривую.

Предикторы в логистической регрессии — это независимые переменные, которые используются для прогнозирования вероятности возникновения некоторого события. Эти переменные могут быть числовыми или категориальными и представляют собой факторы, которые потенциально влияют на зависимую переменную (результат).

Типы предикторов

1. Числовые предикторы: Это переменные, которые имеют числовые значения, такие как возраст, доход или уровень холестерина. Они могут быть непрерывными или дискретными.

2. Категориальные предикторы: Это переменные, которые представляют собой категории, такие как пол, профессия или регион проживания. Категориальные переменные часто кодируются с помощью методов, таких как dummy-кодирование или один-из-n кодирование, чтобы включить их в модель.

3. Бинарные предикторы: Это специальный тип категориальных предикторов, которые могут принимать только два значения, например, «да» или «нет».

Роль предикторов

- Влияние на результат: Каждый предиктор может иметь положительное или отрицательное влияние на вероятность события. Коэффициенты в модели показывают, как изменение предиктора влияет на лог-вероятность события.

- Контроль за другими переменными: Включение нескольких предикторов позволяет контролировать влияние каждого фактора на результат, учитывая эффекты других переменных.

Примеры предикторов

- Возраст и доход в модели кредитного скоринга.

- Уровень холестерина и артериальное давление в модели прогнозирования сердечно-сосудистых заболеваний.

Выбор предикторов

Выбор предикторов для включения в модель логистической регрессии является важным шагом. Он может основываться на теоретических знаниях, экспертном мнении или статистических методах, таких как степwise регрессия или лассо-регуляризация.

Ковариата в логистической регрессии — это независимая переменная, которая включается в модель для контроля ее влияния на зависимую переменную. Ковариаты используются для того, чтобы учесть потенциальные искажения или влияния других факторов на результат, что позволяет более точно оценить эффекты интересующих переменных.

Роль ковариат

1. Контроль за влиянием: Ковариаты помогают контролировать влияние других факторов на зависимую переменную, что важно для получения точных оценок эффектов интересующих переменных.

2. Уменьшение ошибки: Включение ковариат может уменьшить остаточную дисперсию и повысить точность модели, поскольку они объясняют часть вариации в зависимой переменной.

3. Повышение достоверности: Использование ковариат делает результаты модели более достоверными, поскольку они учитывают потенциальные внешние влияния.

Примеры ковариат

- Возраст и пол в модели прогнозирования вероятности заболевания.

- Доход и образование в модели оценки вероятности покупки определенного продукта.

Типы ковариат

1. Числовые ковариаты: Это непрерывные или дискретные числовые переменные, такие как возраст или доход.

2. Категориальные ковариаты: Это переменные, представляющие категории, такие как пол или профессия.

Важность ковариат в логистической регрессии

- Улучшение точности: Ковариаты помогают улучшить точность прогнозов, учитывая влияние других факторов.

- Повышение надежности: Включение ковариат делает результаты модели более надежными и обоснованными.

Выбор ковариат

Выбор ковариат для включения в модель логистической регрессии должен основываться на теоретических знаниях, экспертном мнении или статистических методах. Это важно для того, чтобы избежать переопределения модели и обеспечить ее интерпретируемость.

Разница между ковариатами и предикторами в основном заключается в контексте и цели их использования:

- Предикторы — это переменные, которые напрямую используются для прогнозирования или объяснения результата.

- Ковариаты — это переменные, которые включены в модель для контроля их влияния на результат, часто для того, чтобы более точно оценить эффекты других предикторов.

Пример

Пусть зависимая переменная — доля пациентов, достигших успеха (да/нет). Предикторы: пол, возраст, сопутствующие заболевания. В качестве ковариаты добавляем группу лечения (2 группы).

В данном случае модель логистической регрессии с зависимой переменной "доля пациентов, достигших успеха" и предикторами "пол", "возраст", "сопутствующие заболевания", а также ковариатой "группа лечения" может быть представлена следующим образом:

$$P(Y=1)=\frac{1}{1+e^{-(\beta_0+\beta_1\cdot x_1+\beta_2\cdot x_2+\beta_3\cdot x_3+\beta_4\cdot x_4)}} \quad (224)$$

Здесь:

- Y — зависимая переменная (доля пациентов, достигших успеха, да/нет),

- β_0 — константа (пересечение),

- $\beta_1, \beta_2, \beta_3, \beta_4$ — коэффициенты для пола (x_1), возраста (x_2), сопутствующих заболеваний (x_3) и группы лечения (x_4) соответственно.

Кодирование переменных

- Пол: Обычно кодируется как бинарная переменная (например, мужчина = 0, женщина = 1).

- Возраст: Может быть использован как непрерывная переменная.

- Сопутствующие заболевания: Может быть представлено как бинарная переменная (да/нет) или как счетная переменная (количество заболеваний).

- Группа лечения: Поскольку есть две группы, эту переменную можно закодировать как бинарную (например, 0, 1) .

Пример кода Python:

```
import statsmodels.api as sm

# Предполагая, что данные находятся в DataFrame 'data'

X = data[['sex', 'age', 'comorbidities', 'treatment_group']]

y = data['success']

# Добавление константы (пересечения)

X = sm.add_constant(X)

# Построение модели

model = sm.Logit(y, X).fit()

# Вывод результатов модели

print(model.summary())
```

5.3. Нейронные сети

В общем случае искусственная нейронная сеть (ИНС) может рассматриваться как нелинейная регрессия со специфическим алгоритмом вычисления выходной функции и своей терминологией. Простейшим примером нейронной сети является так называемый однослойный персептрон — это модель искусственного нейрона, используемая для бинарной классификации [42, 43]. Персептрон вычисляет взвешенную сумму входных сигналов и применяет к ней функцию активации (обычно — пороговую).

Математическая запись:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{225}$$

где:

- x_i — значения входных переменных,

- w_i — соответствующие веса этих переменных,

- b — смещение (bias, свободный член),

- f — функция активации (например, ступенчатая или сигмоида),

- y — итоговый выход персептрона (0 или 1, при пороговой функции активации).

Легко видеть, что веса переменных w_i и смещение b являются в данном случае коэффициентами нелинейной регрессии $y=f(x)$.

Примером функции активации может служить логистическая функция (224).

Простая формула (225) позволяет однослойному персептрону выполнять линейно разделимые задачи — отделять данные одной гиперплоскостью. Для более сложных задач требуется использование многослойных структур, в которых выходы одного слоя могут использоваться в качестве входов следующего слоя.

5.3.1.Архитектура искусственной нейронной сети

Архитектура искусственной нейронной сети — это организационная структура нейронов, их взаимосвязей и способов обработки информации внутри сети. Она определяет, как данные поступают на вход, как преобразуются внутри сети и каким образом формируется итоговый результат.

Базовые компоненты архитектуры:

- Входной слой:

Слой, в котором каждый нейрон получает один из входных сигналов системы. Количество нейронов соответствует размерности входных данных.

- Скрытые слои:

Один или несколько промежуточных слоёв, через которые проходит сигнал. Скрытые слои выполняют сложные преобразования входной информации, что позволяет сети находить и обрабатывать сложные взаимосвязи. Глубина (количество скрытых слоёв) и ширина (число нейронов в каждом из них) определяют мощность и универсальность модели. Сети с большим числом скрытых слоёв называют глубокими нейронными сетями (Deep Neural Networks, DNN).

- Выходной слой:

Последний слой сети, формирующий итоговый результат обработки информации. Величина и состав выходного слоя задаются специфическими задачами (классификация, регрессия и др.).

Взаимосвязи и параметры:

- Веса связей (синапсы):

Регулируемые параметры, которые определяют силу связи между отдельными нейронами. Изменение весов в процессе обучения позволяет нейронной сети строить требуемые зависимости между входами и выходом.

- Функции активации:

Специальные математические функции, используемые для вычисления выхода каждого нейрона на основе взвешенной суммы его входов. Наиболее распространены ReLU (Rectified Linear Unit) $f(x)=max(0,x)$ (рис. 32), сигмоида (рис. 33), softmax (рис. 34), и другие.

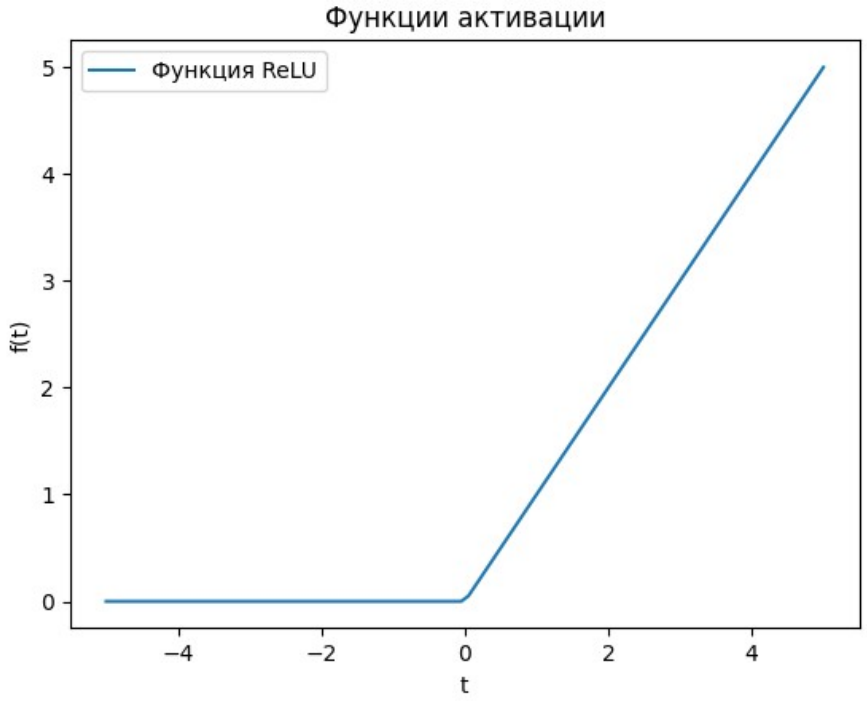

Рисунок 32 - График функции ReLU

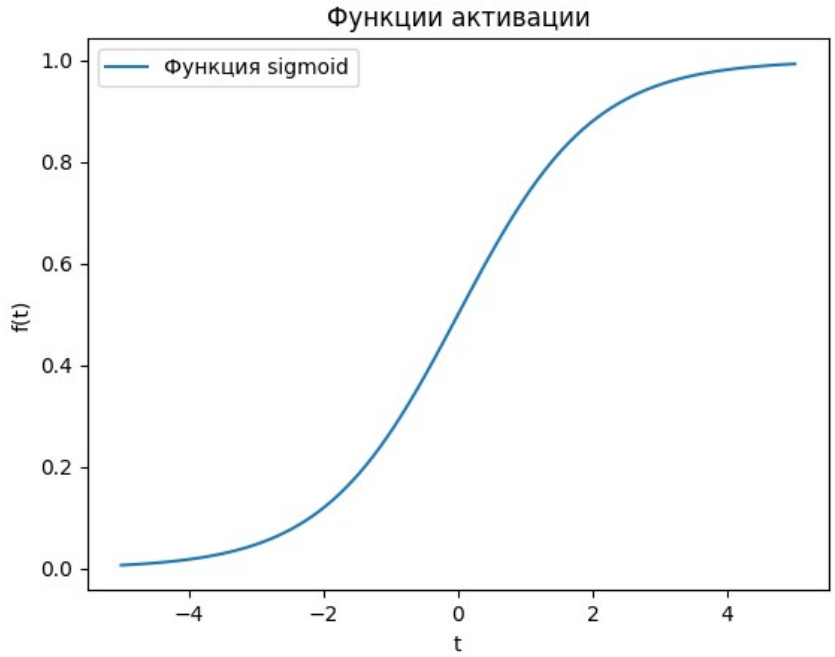

Рисунок 33 - Логистическая функция активации (сигмоида)

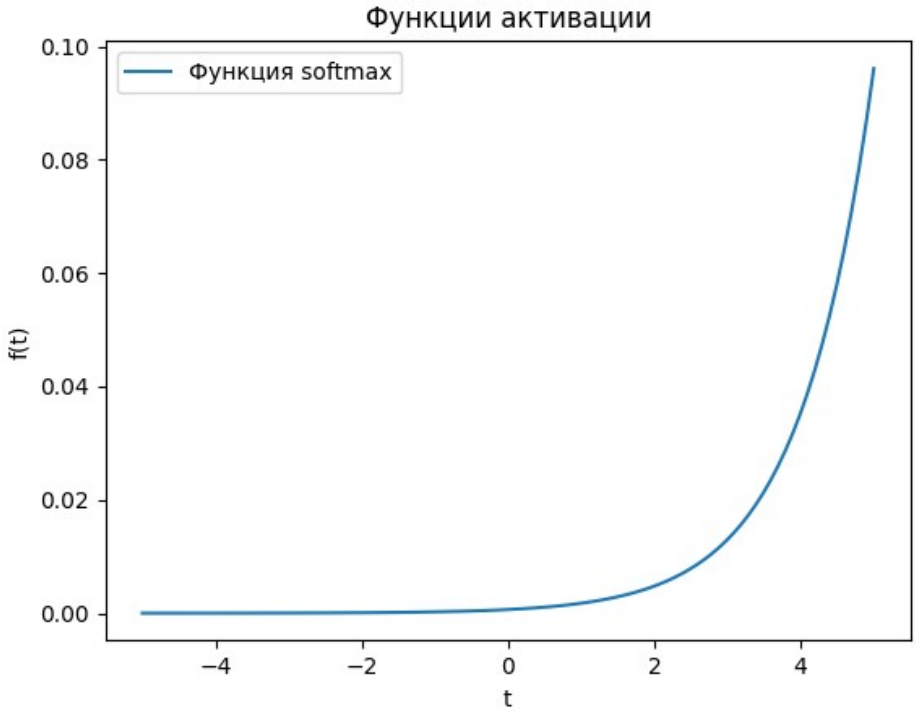

Рисунок 34 - Функция активации softmax

Код программы для отрисовки графиков функций активации:

```
import numpy as np
import matplotlib.pyplot as plt
#Функции активации
def relu(t):
    return np.maximum(t,0)
def softmax(t):
    out = np.exp(t)
    return out / np.sum(out)
def sigmoid(t):
    return 1/(1+np.exp(-t))
t=np.linspace(-5,5,100)
type_fun=['ReLU','softmax','sigmoid']
funcs=[relu(t),softmax(t),sigmoid(t)]
for  i in range(len(funcs)):
    plt.plot(t, funcs[i], label=f'Функция {type_fun[i]}')
    plt.xlabel('t')
    plt.ylabel('f(t)')
    plt.legend()
    plt.title(f'Функции активации')
    plt.show()
```

Основные типы архитектур:

- Прямые нейронные сети (Feedforward Neural Networks):

Информация движется только в одном направлении — от входа к выходу. Такие сети активно применяются для задач распознавания образов, регрессии.

- Сверточные нейронные сети (Convolutional Neural Networks, CNN):

Используют специальные слои для обработки пространственной структуры данных — особенно полезны для анализа изображений и видео.

- Рекуррентные нейронные сети (Recurrent Neural Networks, RNN):

Содержат циклические связи, за счёт чего способны обрабатывать последовательности и временные ряды (например, текст, речь).

- Специализированные и гибридные архитектуры:

Генеративно-состязательные сети (GAN), радиально-базисные сети, самоорганизующиеся карты Кохонена, и другие позволяют решать специфические задачи анализа, генерации и структурирования данных.

Таким образом, архитектура нейронной сети — это совокупность слоёв, нейронов, их связей и функций активации, которые определяют способность ИНС находить сложные закономерности во входных данных и применять их для решения прикладных задач.

Наиболее простую структуру имеет сеть прямого распространения. Математическая формула полносвязного многослойного персептрона (MLP) для слоя с номером l и нейрона с индексом j записывается так:

$$h_j^{(l)} = f\left(\sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} h_i^{(l-1)} + b_j^{(l)}\right), \tag{226}$$

где:

- $h_j^{(l)}$ — выход j-го нейрона слоя l;

- f — функция активации (например, ReLU, сигмоида или tanh);

- $w_{ji}^{(l)}$ — вес связи от i -го нейрона предыдущего слоя ($l-1$) к j -му нейрону слоя l ;

- $h_i^{(l-1)}$ — выход i -го нейрона предыдущего слоя ($h_i^{(0)}$ — входной вектор, т.е. сами признаки);

- $b_j^{(l)}$ — смещение (bias) для j -го нейрона слоя l ;

- n_{l-1} — число нейронов в предыдущем слое.

Для построения всей сети это выражение рекурсивно применяется ко всем слоям (скрытым и выходным). Входной слой не выполняет вычислений, а только передаёт значения на первый скрытый слой.

Для полного многослойного персептрона с L слоями:

- на каждом скрытом и выходном слое применяется та же формула;

- на выходе получается вектор значений (или одна переменная для бинарной классификации).

Обобщённо:

$$\boldsymbol{h}^{(l)} = f\left(\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{227}$$

где жирным записан вектор выходов и веса слоя.

Эта формализация позволяет строить любые глубокие полносвязные нейронные сети прямого распространения.

В архитектуре нейронной сети выделяют следующие основные виды нейронов:

- Входные нейроны:

Получают исходные данные, не выполняя каких-либо вычислений, а лишь передают значения на следующий слой.

- Скрытые нейроны:

Находятся на промежуточных слоях, осуществляют преобразование и вычисление сложных внутренних признаков на основе входных данных. Именно они отвечают за большинство операций по извлечению признаков и обработке информации.

- Выходные нейроны:

Формируют итоговый результат работы сети, например, вероятность принадлежности к классу, численное значение или вектор признаков.

- Нейроны смещения (bias):

Отдельный тип нейронов, который не получает входных данных, а всегда выдаёт заданную константу и позволяет сети лучше приспосабливаться к данным (подобно свободному члену в уравнении регрессии).

Таким образом, разнообразие функций активации и видов нейронов позволяет конструировать гибкие нейросетевые архитектуры для самых различных задач распознавания, прогноза и анализа данных.

5.3.2. Обучение нейронных сетей

Обучение искусственных нейронных сетей заключается в подборе оптимальных весовых коэффициентов для связей между нейронами с целью минимизации ошибки предсказания. Эффективное обучение современных многослойных сетей строится на двух ключевых этапах: прямом распространении (forward propagation) и обратном распространении ошибки (backpropagation).

Прямое распространение

На этапе прямого распространения входные данные подаются на входной слой сети, проходят через все скрытые слои, где преобразуются с помощью взвешенных сумм и функций активации, и, наконец, формируют выход. Этот процесс можно описать как последовательное преобразование векторов активаций от слоя к слою. Веса сети остаются фиксированными: происходит только вычисление выхода сети для заданного входа.

Обратное распространение ошибки

После вычисления выхода сети оценивается ошибка (функция потерь) — разница между полученным значением и целевым (истинным) результатом.

Алгоритм обратного распространения ошибки включает следующие шаги:

1. Вычисление ошибки:

На выходном слое формируется вектор ошибок, который выражает, насколько текущее предсказание отклоняется от правильного ответа.

2. Распределение ошибки по слоям (backpropagation):

Ошибка последовательно распространяется от выходного слоя к входным слоям, вычисляя вклад каждого веса в итоговую ошибку. Благодаря методу дифференцирования (используется правило цепочки для сложных функций) вычисляются частные производные функции потерь по каждому весу сети.

3. Обновление весов:

Каждый вес корректируется на небольшое значение, пропорциональное величине ошибки и производной функции активации по входу (градиенту):

$$w_{new} = w_{old} - \eta \frac{\partial C}{\partial w} \tag{228}$$

где C — функция потерь, η — скорость обучения, $\frac{\partial C}{\partial w}$ — градиент ошибки по весу.

Этот процесс повторяется для каждой выборки обучающего набора, обычно многократно (эпохи), пока ошибка не станет достаточно малой либо не будет достигнут критерий останова.

Математические выражения наиболее часто применяемых функций потерь в нейронных сетях следующие:

1. Среднеквадратичная ошибка (Mean Squared Error, MSE):

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{229}$$

где y_i — истинное значение, $\hat{y}_i$ — предсказание модели, N — число наблюдений. Используется, как правило, в задачах регрессии.

2. Средняя абсолютная ошибка (Mean Absolute Error, MAE):

$$MAE = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{230}$$

Также применяется для задач регрессии, менее чувствительна к выбросам, чем MSE.

3. Кросс-энтропия (Cross Entropy Loss):

- Для бинарной классификации (Binary Cross Entropy):

$$L=-\frac{1}{N}\sum_{i=1}^{N}\left[y_i\log\left(\hat{y}_i\right)+\left(1-y_i\right)\log\left(1-\hat{y}_i\right)\right] \tag{231}$$

- Для многоклассовой классификации (Categorical Cross Entropy):

$$L=-\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}y_{ik}\log\left(\hat{y_{ik}}\right) \tag{232}$$

где K — число классов, y_{ik} — индикатор принадлежности к классу k, $\hat{y_{ik}}$ — вероятность по модели для класса k.

4. Функции потерь для специальных задач:

- Экспоненциальная функция потерь (экспоненциальный Boosting/AdaBoost):

$$L=\sum_{i=1}^{N}\exp\left(-y_i f\left(x_i\right)\right) \tag{233}$$

- Jaccard или IoU loss (для задач сегментации):

$$L=1-\frac{\left|Y\cap\hat{Y}\right|}{\left|Y\cup\hat{Y}\right|}\ . \tag{234}$$

где Y — истинная маска, $\hat{Y}$ — предсказанная маска.

Выбор конкретной функции потерь зависит от типа задачи (регрессия, бинарная или многоклассовая классификация, сегментация и др.), структуры данных и желаемых свойств обучаемой модели.

Совместное применение прямого распространения и обратного распространения ошибки позволило получить эффективные алгоритмы обучения глубоких нейронных сетей, обеспечивающих высокое качество решения сложных задач распознавания, прогнозирования и классификации.

Нейронные сети пригодны для универсальной аппроксимации нелинейных функций: могут моделировать сложнейшие и заранее неизвестные зависимости между многими переменными, включая многоуровневые и вложенные зависимости. Теоретически, сеть с достаточным числом нейронов может аппроксимировать любую непрерывную функцию. Однако избыточно сложные структуры могут привести к переобучению и потере обобщающей способности.

В качестве примера рассмотрим программу аппроксимации экспериментальных данных по влиянию параметров плазменного напыления специального покрытия металла на толщину покрытия, представленных в работе [44]. Атмосферное плазменное напыление (APS) - это специальный процесс, где конечный результат (свойства покрытия) косвенно связан с выбором рабочих параметров. Параметры напыления, изученные в цитированной работе делятся на 5 категорий: параметры исходного материала, энергетические параметры, параметры подачи материала, кинематические параметры и параметры подложки/покрытия. Всего датасет, представленный в данной работе включал 9 независимых параметров и отклик — толщину получаемого покрытия. Программа аппроксимации с использованием ИНС выполняет следующие шаги:

1. Загружает данные из файла 'data.csv'.

2. Определяет архитектуру сети (количество входов, нейронов в скрытом слое и выходов).

3. Определяет функцию активации ReLU и ее производную.

4. Нормализует входные данные и целевую переменную.

5. Инициализирует веса и смещения.

6. Определяет функции прямого и обратного распространения.

7. Выполняет обучение сети методом градиентного спуска.

8. Строит график ошибки в процессе обучения.

9. Строит график корреляции между предсказанными и фактическими значениями.

10. Вычисляет и выводит коэффициент детерминации модели.

Эта реализация использует только numpy для вычислений, что делает ее более прозрачной и легкой для понимания, чем версия с использованием специализированных пакетов TensorFlow и Scikit-learn.

Код программы:

```
import numpy as np
import matplotlib.pyplot as plt
import json
# Загрузка данных
datafile='data.csv'
data = np.loadtxt(datafile)
X = data[:, :-1]
y = data[:, -1].reshape(-1, 1)
# Параметры сети
input_size = X.shape[1]
hidden_size = 64
output_size = 1
# Функции активации
def relu(x):
    return np.maximum(0, x)
def relu_deriv(x):
    return (x > 0).astype(float)
# Нормализация данных
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)
```

```
y_mean = y.mean()
y_std = y.std()
X = (X - X_mean) / X_std
y = (y - y_mean) / y_std
# Инициализация весов и смещений
np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size) * np.sqrt(2. / input_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size) * np.sqrt(2. / hidden_size)
b2 = np.zeros((1, output_size))
# Параметры обучения
ALPHA = 0.001
NUM_EPOCHS = 10000
# Функции прямого и обратного распространения
def forward(X):
    z1 = np.dot(X, W1) + b1
    a1 = relu(z1)
    z2 = np.dot(a1, W2) + b2
    return z1, a1, z2
def backward(X, y, z1, a1, z2):
    m = X.shape[0]
    dz2 = z2 - y
    dW2 = (1/m) * np.dot(a1.T, dz2)
```

```
    db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)
    dz1 = np.dot(dz2, W2.T) * relu_deriv(z1)
    dW1 = (1/m) * np.dot(X.T, dz1)
    db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)
    return dW1, db1, dW2, db2
# Обучение
loss_arr = []
for epoch in range(NUM_EPOCHS):
    z1, a1, z2 = forward(X)
    loss = np.mean((z2 - y) ** 2)
    loss_arr.append(loss)
    dW1, db1, dW2, db2 = backward(X, y, z1, a1, z2)
    W1 -= ALPHA * dW1
    b1 -= ALPHA * db1
    W2 -= ALPHA * dW2
    b2 -= ALPHA * db2
    if epoch % 1000 == 0:
        print(f"Эпохи {epoch}, Ошибка: {loss}")
# Предсказание
def predict(X):
    _, _, z2 = forward(X)
    return z2
```

```
# График ошибки
plt.plot(loss_arr)
plt.title('Изменение ошибки с увеличением числа итераций')
plt.xlabel('Эпохи')
plt.ylabel('Ошибка')
plt.show()
# График корреляции
y_pred = predict(X)
plt.scatter(y, y_pred)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.title('Сравнение предсказанных и фактических данных')
plt.xlabel('Фактические данные')
plt.ylabel('Предсказанные по модели')
plt.show()
# Коэффициент детерминации
r2 = 1 - (np.sum((y - y_pred)**2) / np.sum((y - y.mean())**2))
print(f"Коэффициент детерминации R^2: {r2}")
# Сохранение весов и параметров нормализации в JSON файл
model_data = {
    "W1": W1.tolist(),
    "b1": b1.tolist(),
    "W2": W2.tolist(),
    "b2": b2.tolist(),
```

```
    "X_mean": X_mean.tolist(),
    "X_std": X_std.tolist(),
    "y_mean": float(y_mean),
    "y_std": float(y_std)
}
with open("model_data.json", "w") as f:
    json.dump(model_data, f)
print("Параметры модели сохранены в файле model_data.json")
```

Результаты работы программы:

```
Эпохи 0, Ошибка: 1.7351142679769729
Эпохи 1000, Ошибка: 0.04017696943402169
Эпохи 2000, Ошибка: 0.023128394034939423
Эпохи 3000, Ошибка: 0.020734524756894995
Эпохи 4000, Ошибка: 0.02001471754723845
Эпохи 5000, Ошибка: 0.01961820351921431
Эпохи 6000, Ошибка: 0.01931243200351018
Эпохи 7000, Ошибка: 0.019044053183167882
Эпохи 8000, Ошибка: 0.018800993264146238
Эпохи 9000, Ошибка: 0.01857182524811518
Коэффициент детерминации R^2: 0.9816425657538883
Параметры модели сохранены в файле model_data.json
```

Быстрое уменьшение ошибки и близкое к единице значение коэффициента детерминации модели свидетельствуют о высокой точности аппроксимации. Это иллюстрируется также графиками, представленными на рисунках 35,36.

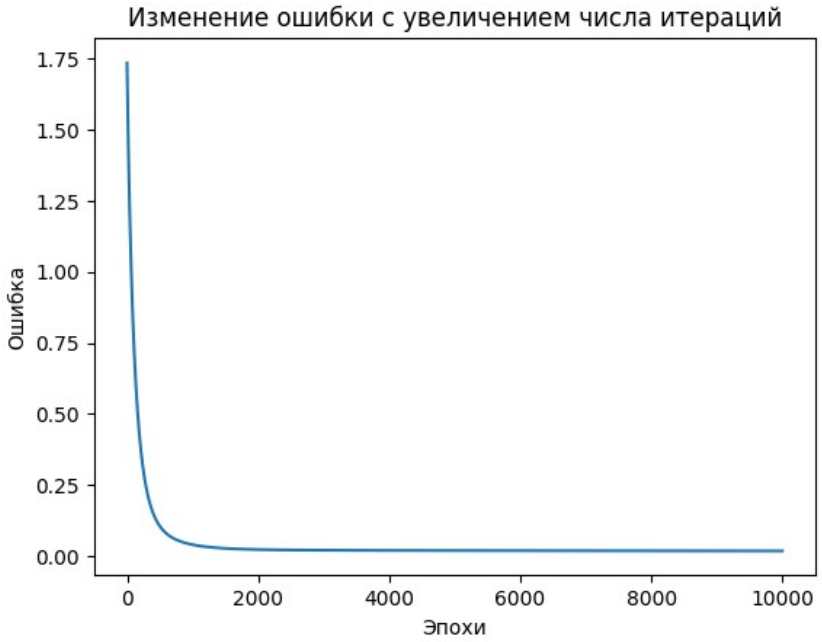

Рисунок 35 - Кривая обучения ИНС

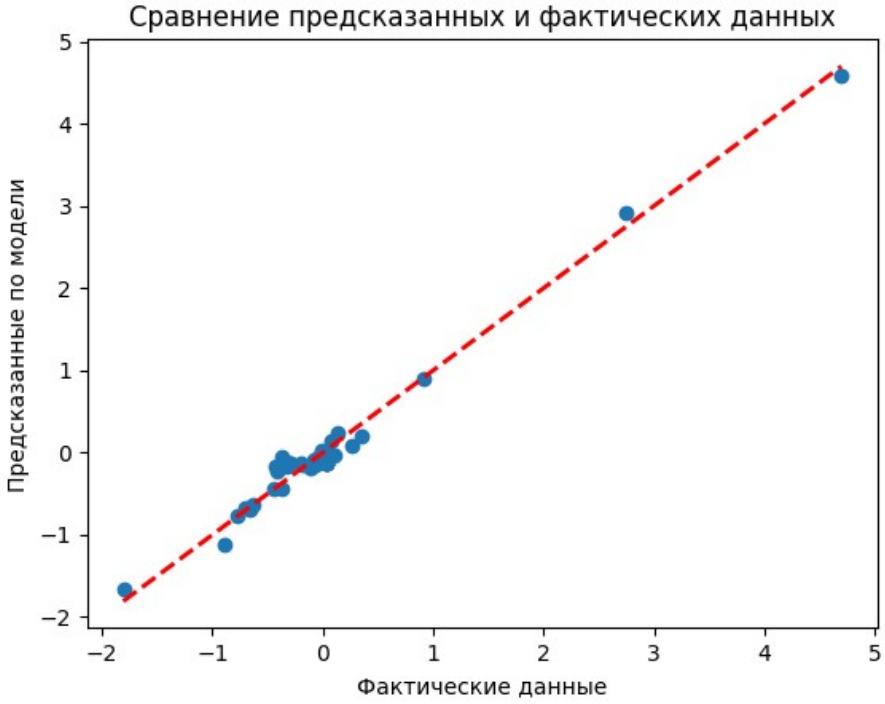

Рисунок 36 - Сравнение фактических значений в выборке с результатами аппроксимации ИНС

5.4. Сравнение подходов: нелинейная регрессия и нейронные сети

В таблице 14 представлены сильные и слабые стороны двух рассмотренных подходов.

Таблица 14. Сравнение нелинейной регрессии и нейронных сетей

Признак	Нелинейная регрессия	Нейронные сети
Интерпретируемость	Высокая: чёткий смысл параметров, легко анализировать вклад переменных	Низкая: модель работает как «чёрный ящик», параметры слабоинтерпретируемы
Область применимости	Подходит для задач с известной или предполагаемой формой зависимости	Идеальна для задач со сложной, неизвестной структурой данных
Требования к объему данных	Эффективна при небольших объемах и ограниченном числе переменных	Требует большого объема и разнообразия данных для качественного обучения
Возможности аппроксимации	Ограничены функциональным видом модели	Универсальны для непрерывных нелинейных зависимостей
Число параметров	Обычно мало — легко управлять и контролировать	Может иметь тысячи и миллионы параметров
Риск переобучения	Обычно невысок, хорошо диагностируется	Высокий при недостатке данных или неправильно выбранной архитектуре
Численные затраты	Минимальны, аналитические решения доступны для ряда моделей	Вычислительно затратны, требуют существенных ресурсов при обучении
Устойчивость к шуму	Зависит от модели и метода оценки, часто ограничена	Высокая устойчивость к зашумленным, неполным данным

Критерии выбора метода для различных задач моделирования

- Цель исследования:

Для задач, где важнее всего интерпретируемость, объяснение результата и анализ вклада переменных, чаще выбирают нелинейную регрессию. Когда требуется максимально точно аппроксимировать сложную зависимость или предсказать результат при неизвестной природе связи — используют нейронные сети.

- Характер данных:

Если структура данных известна или близка к стандартным аналитическим функциям, регрессия показывает отличные результаты. При наличии множества факторов, скрытых и сложных взаимосвязей, преимуществом обладают нейросети.

- Объем и качество обучающей выборки:

Нейронные сети требуют значительно большего объёма данных для обучения, а также тщательной настройки параметров и архитектуры.

- Требования к скорости и вычислительным ресурсам:

Для простых и срочных задач подойдут методы регрессии; если допускается длительное обучение и используются высокопроизводительные вычисления — нейросети.

Вывод:

Оба подхода актуальны для задач моделирования систем, однако выбор метода всегда должен соотноситься с особенностями предметной области, доступными данными и требованиями к результату: от объяснимости до максимального качества аппроксимации.

Глава 6 Имитационное моделирование систем на основе теории массового обслуживания

Имитационное моделирование систем на основе теории сетей массового обслуживания занимает важное место в анализе сложных дискретных объектов со стохастическим характером функционирования [45, 46]. Современные системы массового обслуживания, такие как транспортные, информационно-коммуникационные, производственные и сервисные сети, обладают высокой сложностью, многовариантностью потоков и требуют разработки эффективных методов исследования их поведения во времени.

Аналитические методы теории массового обслуживания позволяют рассчитывать важнейшие характеристики работы систем — среднее число заявок в очереди, время ожидания, интенсивности потоков и другие параметры. Однако с ростом сложности моделируемых сетей аналитические решения становятся громоздкими либо невозможными, особенно при наличии нескольких каналов обслуживания, разветвленных потоков, ограничений на ресурсы и вероятностных отказов.

Имитационное моделирование предоставляет инструменты для воспроизведения поведения систем массового обслуживания на вычислительных моделях в соответствии с их логической структурой, позволяя анализировать их функционирование в широком диапазоне сценариев. Такой подход даёт возможность получить статистические оценки эффективности работы системы, выявлять «узкие места», анализировать влияние параметров и строить оптимальные варианты её организации.

В рамках этого раздела рассматриваются основные понятия теории сетей массового обслуживания, подходы к построению имитационных моделей, современные инструменты моделирования с использованием Python, а также методы анализа получаемых результатов. Особое внимание уделяется практическим аспектам разработки и исследования моделей сетей массового обслуживания, что позволяет связать теоретические знания с реальными задачами анализа и оптимизации систем различной природы.

6.1 Основные положения теории массового обслуживания (ТМО)

Теория массового обслуживания (ТМО) — раздел теории вероятностей, который занимается исследованием показателей производительности систем, предназначенных для обработки поступающих в них заявок на обслуживание. Предметом ТМО является установление зависимости между основными характеристиками системы обслуживания с целью улучшения управления системами.

Теория потока однородных событий, которая легла в основу ТМО, была разработана советским математиком А.Я. Хинчиным. Первые задачи ТМО были рассмотрены сотрудником Копенгагенской телефонной компании Агнером Эрлангом в период между 1908 и 1922 годами.

Основные задачи теории массового обслуживания включают:

1. Построение математической модели системы массового обслуживания и расчет её основных характеристик

2. Установление зависимости эффективности работы системы от её организации

3. Решение различных оптимизационных задач, связанных с функционированием системы массового обслуживания

4. Выработка рекомендаций по рациональному построению системы массового обслуживания для обеспечения высокой эффективности её функционирования

Все задачи ТМО носят оптимизационный характер и направлены на определение такого варианта работы системы, при котором будет обеспечен минимум суммарных затрат.

6.1.1. Основные элементы системы массового обслуживания (СМО)

Системой массового обслуживания (СМО) называется любая система, предназначенная для обслуживания какого-либо потока заявок. Примерами систем массового обслуживания могут служить:

- посты технического обслуживания автомобилей;
- персональные компьютеры, обслуживающие поступающие заявки или требования на решение тех или иных задач;

- аудиторские фирмы;
- банки;
- телефонные станции.

В СМО обслуживаемый объект называют *требованием*. В общем случае под требованием обычно понимают запрос на удовлетворение некоторой потребности, например, разговор с абонентом, посадка самолета, покупка билета, получение материалов на складе. Средства, обслуживающие требования, называются обслуживающими устройствами или каналами обслуживания.

Предметом теории массового обслуживания является установление зависимости между факторами, определяющими функциональные возможности системы массового обслуживания, и эффективностью ее функционирования.

Структурные компоненты

Основными структурными элементами любой СМО являются:

1. Источник требований — устройство или группа устройств, от которых поступают требования в обслуживающую систему.

2. Входящий поток заявок — последовательность появления во времени требований, поступающих от источника.

3. Очередь — совокупность требований, которые не могут быть сразу удовлетворены.

4. Каналы обслуживания — обслуживающие устройства, выполняющие операции обслуживания.

5. Выходящий поток обслуженных требований — поток требований, выходящий из обслуживающего устройства.

Основные параметры СМО

Интенсивность потока заявок (λ) — среднее число заявок, поступающих в систему в единицу времени.

Интенсивность обслуживания (μ) — величина, обратная среднему времени обслуживания, или число заявок, которое может обслужить система в единицу времени.

Приведенная интенсивность потока заявок (ρ) — отношение интенсивности поступления заявок к интенсивности обслуживания: $\rho = \lambda/\mu$.

Классификация систем массового обслуживания

По количеству каналов обслуживания:

- Одноканальные системы — имеют один канал обслуживания
- Многоканальные системы — имеют несколько каналов обслуживания

По дисциплине обслуживания:

- СМО с отказами — заявка, поступившая в момент занятости всех каналов, получает отказ и покидает систему.
- СМО с ожиданием (неограниченным ожиданием) — заявка становится в очередь и ожидает освобождения канала.

- СМО смешанного типа (с ограниченным ожиданием) — на пребывание заявки в очереди накладываются ограничения по длине очереди или времени ожидания.

По взаимному расположению каналов:

- Системы с параллельным расположением каналов — обслуживание заявок может вести любой свободный канал
- Системы с последовательным расположением каналов — каждый последующий канал может приступить к обслуживанию только после предыдущего

Классификация по Кендаллу-Ли

При классификации по Кендаллу-Ли системы массового обслуживания (СМО) обозначаются в виде записи: A/B/C:D/E/F,

где:

- A — распределение интервалов времени между поступлениями требований (например, M — пуассоновский поток, D — детерминированный, G — произвольный),

- B — распределение времени обслуживания (M — экспоненциальное, D — детерминированное, G — произвольное),

- C — число параллельных каналов обслуживания,

- D — дисциплина обслуживания (FIFO, LIFO, SIRO и др.),

- E — максимальное число требований в системе,

- F — размер источника (максимальное число клиентов, которые могут появиться).

Примеры обозначений:

- M/M/1 — одноканальная СМО с пуассоновским потоком заявок, экспоненциальным обслуживанием, одним каналом, бесконечной очередью и дисциплиной FIFO.

- M/G/2 — двухканальная СМО с пуассоновским потоком, произвольным распределением времени обслуживания и двумя каналами.

- M/M/c — система с с каналами, пуассоновским входящим потоком, экспоненциальным обслуживанием.

- M/M/1:K — одноканальная система с ограничением на максимальное число заявок в системе (K).

- D/M/1 — раз в фиксированные интервалы (детерминированно) поступают заявки, обслуживание экспоненциальное.

- M/D/1 — пуассоновский входящий поток, обслуживание детерминированное, один канал.

- G/G/1 — система с произвольными интервалами между поступлениями и временем обслуживания, один канал.

В ряде случаев в обозначении указывают только первые три параметра, если остальные параметры считаются стандартными (например, неограниченная очередь, дисциплина обслуживания FIFO и неограниченное число источников).

Потоки событий в ТМО

1. Простейший (пуассоновский) поток

Простейший поток — поток событий, обладающий свойствами стационарности, ординарности и отсутствия последействия:

Стационарность — вероятность появления k событий на любом промежутке времени зависит только от числа k и от длительности t промежутка и не зависит от начала его отсчёта.

Ординарность — вероятность одновременного появления двух и более событий равна нулю.

Отсутствие последействия — вероятность появления k событий на любом промежутке времени не зависит от того, появлялись или не появлялись события в предыдущие моменты времени.

Для простейшего потока вероятность появления k событий за время t определяется формулой Пуассона:

$$P_t(k)=(\lambda t)^k \times e^{(-\lambda t)}/k! \qquad (235)$$

2. Регулярный поток (детерминированный поток)

В регулярном потоке события происходят строго через одинаковые интервалы T, то есть моменты наступления событий: $t_k=kT$, где $k=1,2,3,....$

- Интервал между событиями фиксирован:

$$P(\textit{интервал между событиями}=T)=1$$

- Число событий за время t :

$$N(t)=\frac{t}{T}$$

3. Нестационарный пуассоновский поток

В этом потоке интенсивность поступления событий меняется со временем (λ зависит от времени):

- Вероятность появления k событий на интервале $[0,t]$:

$$P_t(k)=\frac{\Lambda(t)^k}{k!}e^{-\Lambda(t)}, \tag{236}$$

где $\Lambda(t)=\int_0^t \Lambda(s)ds$, $\Lambda(s)$ — мгновенная интенсивность потока в момент времени s .

4. Пакетный (неординарный) поток

Допустим, что события приходят пакетами (по M событий одновременно), а моменты поступления пакетов подчиняются пуассоновскому закону с интенсивностью v :

- Число пакетов за время t распределено по Пуассону:

$$P_t(k)=\frac{(vt)^k}{k!}e^{-vt} \tag{237}$$

- Общее число событий за время t : $N(t)=M\times K(t)$, где $K(t)$ — число пакетов за интервал.

Эти формулы различают основные примеры потоков событий и могут быть использованы для моделирования в задачах анализа и имитационной оценки систем обслуживания.

Распределения времени обслуживания

1. Экспоненциальное распределение

Экспоненциальное распределение широко используется в ТМО для моделирования времени обслуживания заявок. Плотность вероятности имеет вид:

$$f(x)=\lambda e^{(-\lambda x)}, \quad для\ x\geq 0 \tag{238}$$

Экспоненциальное распределение применяется для описания:

- Времени между поступлениями заявок в систему

- Продолжительности обслуживания

- Времени между отказами оборудования

- Интервалов времени между поломками

2. Детерминированное распределение (распределение D)

- Время обслуживания фиксировано и всегда равно одной и той же величине T .

- Формула плотности вероятности:

$$f(t)=\delta(t-T) \tag{239}$$

где $\delta(x)$ — дельта-функция.

3. Распределение Эрланга (E_k или Erlang, k-порядка)

- Моделирует время обслуживания как сумму k независимых экспоненциальных величин с параметром λ .

- Формула плотности вероятности:

$$f(t)=\frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!}, \quad t\geq 0 \tag{240}$$

4. Равномерное распределение (Uniform)

- Время обслуживания распределено равномерно на отрезке [a, b].

- Формула плотности вероятности:

$$f(t)=\begin{cases} \frac{1}{b-a}, & a\leq t\leq b \\ 0, & \textit{иначе} \end{cases} . \tag{241}$$

5. Гамма-распределение (Gamma)

- Обобщение распределения Эрланга для вещественного параметра формы k .

- Формула плотности вероятности:

$$f(t)=\frac{\lambda^k t^{k-1} e^{-\lambda t}}{\Gamma(k)}\ , \qquad (242)$$

где $\Gamma(k)$ — гамма-функция.

6. Нормальное (гауссовское) распределение

- Время обслуживания подчиняется закону нормального распределения с параметрами μ (среднее) и σ^2 (дисперсия).

- Формула плотности вероятности:

$$f(t)=\frac{1}{\sqrt{2\pi\sigma^2}},e^{-\frac{(t-\mu)^2}{2\sigma^2}} \qquad (243)$$

7. Произвольное распределение (G)

- В общем случае в моделях могут применяться любые (G — general) законы распределения времени обслуживания, например, логнормальное, гиперэкспоненциальное и пр.

Эти распределения позволяют моделировать разнообразные реальные процессы, выходящие за рамки случаев с экспоненциальным временем обслуживания.

Дисциплины обслуживания

Бесприоритетные дисциплины:

FIFO/FCFS (First In, First Out / First Come, First Served) — обслуживание в порядке поступления заявок.

LIFO/LCFS (Last In, First Out / Last Come, First Served) — обслуживание в обратном порядке поступления.

SIRO (Service In Random Order) — случайный выбор заявки из очереди на обслуживание.

Приоритетные дисциплины:

SRPT (Shortest Remaining Processing Time) — обслуживание первой заявки с минимальным остаточным временем обслуживания.

LRPT (Longest Remaining Processing Time) — обслуживание первой заявки с максимальным остаточным временем обслуживания.

RR (Round-Robin) — обслуживание в режиме разделения времени с квантами.

Показатели эффективности СМО

Показатели для СМО с отказами

Абсолютная пропускная способность (A) — среднее число заявок, которое может обслужить система за единицу времени.

Относительная пропускная способность (q) — средняя доля поступивших заявок, обслуживаемая системой.

Вероятность обслуживания заявки ($P_{обс}$) — вероятность того, что заявка будет принята к обслуживанию.

Вероятность отказа ($P_{отк}$) — вероятность того, что заявка получит отказ в обслуживании.

Показатели для СМО с ожиданием:

Среднее число заявок в очереди — математическое ожидание длины очереди.

Среднее число обслуживаемых заявок — среднее число заявок, находящихся на обслуживании.

Среднее время ожидания заявки в очереди — математическое ожидание времени пребывания заявки в очереди.

Среднее время пребывания заявки в системе — общее время нахождения заявки в СМО.

Марковские процессы в ТМО

Методология ТМО включает все основные элементы теории случайных процессов: марковские случайные процессы, потоки заявок, граф состояний, системы обыкновенных дифференциальных уравнений для вероятностей состояний.

Марковский процесс — случайный процесс, обладающий свойством отсутствия последействия, когда будущее состояние системы зависит только от настоящего состояния и не зависит от предыстории. Математическое условие марковости (марковское свойство) для случайного процесса X_t формулируется так:

Вероятность того, что процесс примет значение x_n в момент времени t_n , при условии всех предыдущих состояний, зависит только от текущего состояния x_{n-1} в момент t_{n-1} :

$$P\left(X_{t_n}=x_n \middle| X_{t_{n-1}}=x_{n-1}, X_{t_{n-2}}=x_{n-2}, \ldots, X_{t_0}=x_0\right)=P\left(X_{t_n}=x_n \middle| X_{t_{n-1}}=x_{n-1}\right) . \quad (244)$$

В ТМО часто используется граф состояний, изображающий процесс гибели и размножения, где каждое промежуточное состояние связано прямой и обратной связью с каждым соседним состоянием.

Основные положения ТМО обеспечивают теоретическую базу для анализа и оптимизации систем массового обслуживания различной сложности, от простейших одноканальных систем до сложных сетей массового обслуживания.

6.1.2. Математические модели СМО

Математические модели в теории массового обслуживания служат инструментом для анализа функционирования систем обслуживания и прогнозирования их характеристик. Эти модели отражают случайную природу поступления, ожидания и обслуживания заявок, а также позволяют вычислять ключевые показатели эффективности системы.

Каждой модели соответствует система уравнений (обыкновенных или разностных), описывающих вероятности нахождения системы в различных состояниях. Решение этих уравнений

позволяет найти вероятности занятости каналов, длины очереди, средней нагрузки и другие характеристики. Ниже приведены расчетные формулы и примеры некоторых изученных в ТМО моделей стационарных систем марковского типа.

Входные параметры:

интенсивность входящего потока заявок λ,

интенсивность выходящего потока обслуженных заявок μ,

приведенная интенсивность потока заявок $\rho = \lambda / \mu$,

число каналов обслуживания n ,

длина очереди m

считаются известными. Для вычисления показателей эффективности СМО используются формулы, приведенные в таблицах 15-17 .

Таблица 15. Одноканальная система с отказами

Наименование показателя	Расчетная формула
Среднее время обслуживания заявки	$\bar{t}_{обс} = \frac{1}{\mu}$
Относительная пропускная способность СМО	$q = \frac{1}{\rho + 1}$
Абсолютная пропускная способность СМО	$A = \lambda q$
Вероятность того, что заявка будет обслужена	$P_{обс} = q$
Вероятность того, что заявка получит отказ	$P_{отк} = 1 - P_{обс}$

Таблица 16. Многоканальная система с отказами

Наименование показателя	Расчетная формула
Вероятность p_n того, что занято $0,1,\dots,n$ каналов, соответственно	$\begin{cases} p_0=\left(\sum\limits_{k=0}^{n}\frac{\rho^k}{k!}\right)^{-1} \\ p_k=\frac{\rho^k}{k!}\cdot p_0 \end{cases}, \quad k=1,2,\dots n$
Относительная пропускная способность СМО	$q=1-\frac{\rho^n}{n!}\cdot p_0$
Абсолютная пропускная способность СМО	$A=\lambda q$
Вероятность того, что заявка будет обслужена	$P_{обс}=q$
Вероятность того, что заявка получит отказ	$P_{отк}=1-P_{обс}$
Среднее число занятых каналов	$\bar{k}=\frac{A}{\mu}$

Таблица 17. Одноканальная система с ограниченной очередью

Наименование показателя	Расчетная формула
Вероятность того, что СМО свободна и может обслужить заявку	$p_0=\begin{cases}\frac{1-\rho}{1-\rho^{m+2}}, \rho\neq 1\\ \frac{1}{m+2}, \rho\neq 1\end{cases}$
Вероятности того, что СМО занята, а в очереди находятся $1,2,\dots,m$ заявок, соответственно	$p_k=\rho^k\cdot p_0\ ,\ \ k=1,2,\dots m+1$
Относительная пропускная способность СМО	$q=1-p_{m+1}$
Абсолютная пропускная способность СМО	$A=\lambda q$
Вероятность того, что заявка будет обслужена	$P_{обс}=q$
Вероятность того, что заявка получит отказ	$P_{отк}=1-P_{обс}$
Среднее число заявок, стоящих в очереди	$\bar{r}=\begin{cases}\frac{\rho^2[1-\rho^m(m+1-m\rho)]}{(1-\rho^{m+2})(1-\rho)}, \rho\neq 1\\ \frac{m(m+1)}{2(m+2)}, \rho=1\end{cases}$
Среднее число заявок в СМО (обслуживаемых и стоящих в очереди)	$\bar{k}=\bar{r}+1-p_0$
Среднее время ожидания заявки в очереди	$\bar{t}_{ож}=\bar{r}/\lambda$
Среднее время пребывания заявки в СМО	$\bar{t}_{СМО}=\frac{\bar{r}}{\lambda}+\frac{q}{\mu}$

Ниже приведен текст программы, в которой реализованы формулы расчета показателей эффективности при заданных параметрах системы для трех рассмотренных выше СМО.

```
import math

def single_channel_loss_system(lmbd, mu):

    # Одноканальная с отказами

    rho = lmbd / mu
```

```
    t_obs = 1 / mu
    q = 1 / (rho + 1)
    A = lmbd * q
    P_obs = q
    P_otk = 1 - P_obs
    return {
        "Среднее время обслуживания заявки": t_obs,
        "Относительная пропускная способность": q,
        "Абсолютная пропускная способность": A,
        "Вероятность обслуживания": P_obs,
        "Вероятность отказа": P_otk
    }
def multi_channel_loss_system(lmbd, mu, n):
    # Многоканальная с отказами (n каналов)
    rho = lmbd / mu
    sum_terms = sum([(rho ** k) / math.factorial(k) for k in range(n + 1)])
    p0 = 1 / sum_terms
    pn = [(rho ** k) / math.factorial(k) * p0 for k in range(n + 1)]
    q = 1 - pn[n]
    A = lmbd * q
    P_obs = q
    P_otk = 1 - P_obs
    k_avg = A / mu
```

```
    return {
        "Вероятности занятости каналов": pn,
        "Относительная пропускная способность": q,
        "Абсолютная пропускная способность": A,
        "Вероятность обслуживания": P_obs,
        "Вероятность отказа": P_otk,
        "Среднее число занятых каналов": k_avg
    }
def single_channel_limited_queue(lmbd, mu, m):
    # Одноканальная с ограниченной очередью
    rho = lmbd / mu
    if rho != 1:
        p0 = (1 - rho) / (1 - rho ** (m + 2))
    else:
        p0 = 1 / (m + 2)
    pk = [rho ** k * p0 for k in range(m + 2)]
    q = 1 - pk[m + 1]
    A = lmbd * q
    P_obs = q
    P_otk = 1 - P_obs
    if rho != 1:
        r_avg = (rho / 2) * (1 - rho ** m * (m + 1 - m * rho)) / ((1 - rho ** (m + 2)) * (1 - rho))
    else:
```

```
        r_avg = m * (m + 1) / (2 * (m + 2))
    k_avg = r_avg + 1 - p0
    t_oj = r_avg / lmbd
    t_smo = r_avg / lmbd + q / mu
    return {
        "Вероятности состояний системы": pk,
        "Относительная пропускная способность": q,
        "Абсолютная пропускная способность": A,
        "Вероятность обслуживания": P_obs,
        "Вероятность отказа": P_otk,
        "Среднее число заявок в очереди": r_avg,
        "Среднее число заявок в системе": k_avg,
        "Среднее время ожидания": t_oj,
        "Среднее время пребывания": t_smo
    }
def main():
    print("Выберите тип системы массового обслуживания:")
    print("1 - Одноканальная система с отказами")
    print("2 - Многоканальная система с отказами")
    print("3 - Одноканальная система с ограниченной очередью")
    choice = int(input("Введите номер варианта: "))
    lmbd = float(input("Введите интенсивность входящего потока заявок λ: "))
    mu = float(input("Введите интенсивность обслуживания μ: "))
```

```
    if choice == 1:
        result = single_channel_loss_system(lmbd, mu)
    elif choice == 2:
        n = int(input("Введите число каналов обслуживания n: "))
        result = multi_channel_loss_system(lmbd, mu, n)
    elif choice == 3:
        m = int(input("Введите длину очереди m: "))
        result = single_channel_limited_queue(lmbd, mu, m)
    else:
        print("Неверный выбор.")
        return
    print("\nРезультаты расчетов:")
    for key, value in result.items():
        if isinstance(value, list):
            print(f"{key}:")
            for i, val in enumerate(value):
                print(f"  {i}: {val:.6f}")
        else:
            print(f"{key}: {value:.6f}")
if __name__ == "__main__":
    main()
```

Пользователь выбирает тип СМО, вводит значения входных параметров, а программа рассчитывает показатели эффективности по формулам из вашего файла. Пример охватывает

одноканальную с отказами, многоканальную с отказами и одноканальную с ограниченной очередью.

Пример вывода результатов:

Выберите тип системы массового обслуживания:

1 - Одноканальная система с отказами

2 - Многоканальная система с отказами

3 - Одноканальная система с ограниченной очередью

Введите номер варианта: 2

Введите интенсивность входящего потока заявок λ: 11

Введите интенсивность обслуживания μ: 0.15

Введите число каналов обслуживания n: 3

Результаты расчетов:

Вероятности занятости каналов:

0: 0.000015

1: 0.001071

2: 0.039259

3: 0.959656

Относительная пропускная способность: 0.040344

Абсолютная пропускная способность: 0.443783

Вероятность обслуживания: 0.040344

Вероятность отказа: 0.959656

Среднее число занятых каналов: 2.958556

Меняя значения параметров для выбранной СМО можно оценить ее эффективность и, при необходимости, подобрать параметры, обеспечивающие повышение эффективности работы системы.

6.1.3. Система уравнений Колмогорова для вероятностей состояний СМО

Система уравнений Колмогорова представляет собой математический аппарат для описания марковских случайных процессов с дискретными состояниями и непрерывным временем, который широко применяется в анализе систем массового обслуживания.

Пусть $p_i(t)$ — вероятность того, что система находится в состоянии S_i в момент времени t .

Формулировка уравнений Колмогорова

Для марковского процесса с дискретными состояниями $S_0, S_1, S_2, \ldots$ и интенсивностями переходов λ_{ij} (из состояния i в состояние j) система уравнений Колмогорова имеет вид:

$$\frac{dp_i(t)}{dt} = \sum_{j \neq i} \lambda_{ji} p_j(t) - p_i(t) \sum_{j \neq i} \lambda_{ij} \tag{245}$$

где:

- $\frac{dp_i(t)}{dt}$ — производная вероятности i -го состояния по времени
- $\lambda_{ji} p_j(t)$ — поток вероятности, входящий в состояние i из состояния j
- $p_i(t) \lambda_{ij}$ — поток вероятности, выходящий из состояния i в состояние j

Правила составления уравнений Колмогорова

Система составляется по размеченному графу состояний СМО согласно следующим правилам:

1. В левой части уравнения стоит производная вероятности рассматриваемого состояния $\frac{dp_i(t)}{dt}$

2. В правой части уравнения:

- Сумма произведений интенсивностей всех потоков, входящих в состояние S_i , на вероятности тех состояний, из которых эти потоки исходят

- Минус произведение вероятности данного состояния $p_i(t)$ на суммарную интенсивность всех потоков, выходящих из этого состояния

3. Нормировочное условие: $\sum_{i=0^{\infty}} p_i(t)=1$

Пример составления системы уравнений

Для трехканальной СМО с состояниями S_0, S_1, S_2 и интенсивностями:

- $\lambda_{01}=2, \lambda_{10}=4$
- $\lambda_{12}=3, \lambda_{21}=2$
- $\lambda_{20}=4$

Система уравнений Колмогорова:

$$\begin{cases} \dfrac{dp_0(t)}{dt}=4\,p_1(t)+4\,p_2(t)-2\,p_0(t) \\ \dfrac{dp_1(t)}{dt}=2\,p_0(t)+2\,p_2(t)-(4+3)\,p_1(t) \\ \dfrac{dp_2(t)}{dt}=3\,p_1(t)-(2+4)\,p_2(t) \end{cases} \quad (246)$$

Стационарный режим и финальные вероятности

В стационарном режиме при $t \to \infty$ вероятности состояний стремятся к постоянным значениям — финальным (предельным) вероятностям p_i .

Поскольку в стационарном режиме $\frac{dp_i(t)}{dt}=0$, система дифференциальных уравнений Колмогорова превращается в систему линейных алгебраических уравнений:

$$\begin{cases} 0=\sum_{j\neq i}\lambda_{ji}p_j-p_i\sum_{j\neq i}\lambda_{ij} \\ \sum_{i=0}^{\infty}p_i=1 \end{cases} \quad (247)$$

Физический смысл финальных вероятностей

Финальная вероятность состояния p_i показывает среднее относительное время пребывания системы в этом состоянии. Например, если $p_0=0,2$, то система в среднем 20% рабочего времени находится в состоянии S_0 .

Условия существования решения

Предельные вероятности существуют, если:

- Число состояний системы конечно

- Из каждого состояния можно перейти в любое другое состояние за конечное число шагов

Решение системы уравнений

Для решения системы уравнений Колмогорова необходимо:

1. Для нестационарного режима: задать начальные условия $p_i(0)$ и проинтегрировать систему дифференциальных уравнений

2. Для стационарного режима: решить систему линейных алгебраических уравнений с учетом нормировочного условия

Система уравнений Колмогорова является фундаментальным инструментом анализа марковских СМО, позволяющим получить все основные характеристики эффективности системы через вероятности её состояний.

Рассмотрим решение задачи, представленной уравнениями (246),(247). В матричном виде систему уравнений Колмогорова для данной задачи можно записать как:

$$\frac{d\,\boldsymbol{p}}{dt} = Q\,\boldsymbol{p}\ , \tag{248}$$

где $Q = \begin{pmatrix} -2 & 4 & 4 \\ 2 & -7 & 2 \\ 0 & 3 & -6 \end{pmatrix}$, $\boldsymbol{p} = (p_0, p_1, p_2)^T$.

Эта линейная система обыкновенных дифференциальных уравнений может быть решена с помощью матричной экспоненты

$$\boldsymbol{p}(t) = \boldsymbol{p}(0)e^{Qt}\ , \tag{249}$$

где $\boldsymbol{p}(0)$ — значение вектора вероятностей в начальный момент времени.

Также можно решить эту систему с использованием стандартных численных методов решения систем обыкновенных дифференциальных уравнений, например, методом Рунге-Кутты.

Ниже приведен текст программы с решением данной задачи

```
import numpy as np
from scipy.linalg import expm, solve
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
# Матрица коэфициентов
Q = np.array([
   [-2,  4,  4],
   [ 2, -7,  2],
   [ 0,  3, -6]
])
p0 = np.array([1, 0, 0])   # Начальное состояние-столбец
```

```
# Временная сетка
t_values = np.linspace(0, 10, 200)
# Решение экспонентой:
# для столбцового вектора состояния нужно expm(Q * t) @ p0
p_exp = np.array([expm(Q * t) @ p0 for t in t_values])
# Решение методом Рунге–Кутта
def kolmogorov_eq(t, p, Q):
    return Q @ p
sol = solve_ivp(
    kolmogorov_eq,
    [t_values[0], t_values[-1]],
    p0, args=(Q,), t_eval=t_values, method='RK45'
)
# Стационарные вероятности: Q p = 0 с нормировкой
A = Q.copy()
A[-1, :] = 1   # Последнее уравнение — сумма вероятностей = 1
b = np.array([0, 0, 1])
steady_state = solve(A, b)
# Печать результатов для нескольких точек во времени
check_indices = [0, 20, 50, 100, 150, 199]
print("► Эволюция вероятностей по времени (матрица-экспоненты):")
for idx in check_indices:
```

```
    t = t_values[idx]
    pt = p_exp[idx]
    print(f"t={t:>4.2f}: P0={pt[0]:.6f}, P1={pt[1]:.6f}, P2={pt[2]:.6f}, sum={pt.sum():.6f}")
print("\n► Эволюция вероятностей (метод Рунге–Кутта):")
for idx in check_indices:
    t = sol.t[idx]
    pvec = sol.y[:, idx]
    print(f"t={t:>4.2f}: P0={pvec[0]:.6f}, P1={pvec[1]:.6f}, P2={pvec[2]:.6f}, sum={pvec.sum():.6f}")
print("\n► Стационарные вероятности (равновесные):")
print(f"P0={steady_state[0]:.6f}, P1={steady_state[1]:.6f}, P2={steady_state[2]:.6f},
sum={steady_state.sum():.6f}")
# Графики
plt.figure(figsize=(10, 6))
plt.plot(t_values, p_exp[:, 0], 'r-',   label=r'$P_0$ (экспонента)')
plt.plot(t_values, p_exp[:, 1], 'g-',   label=r'$P_1$ (экспонента)')
plt.plot(t_values, p_exp[:, 2], 'b-',   label=r'$P_2$ (экспонента)')
plt.plot(sol.t, sol.y[0], 'r--', label=r'$P_0$ (РК)')
plt.plot(sol.t, sol.y[1], 'g--', label=r'$P_1$ (РК)')
plt.plot(sol.t, sol.y[2], 'b--', label=r'$P_2$ (РК)')
plt.axhline(steady_state[0], color='r', linestyle=':', label=r'$P_0^{ст}$')
plt.axhline(steady_state[1], color='g', linestyle=':', label=r'$P_1^{ст}$')
plt.axhline(steady_state[2], color='b', linestyle=':', label=r'$P_2^{ст}$')
plt.xlabel('Время t')
plt.ylabel('Вероятность')
```

```
plt.title('Эволюция вероятностей состояний системы')
plt.legend(loc='right')
plt.grid(True)
plt.tight_layout()
plt.show()
```

В данной программе система уравнений Колмогорова решается двумя методами: с помощью матричной экспоненты и методом Рунге-Кутте с использованием функций expm(Q * t) и solve_ivp , соответственно. Система уравнений стационарных вероятностей (247) решается с помощью функции линейной алгебры solve(A, b).

Ниже приведен текстовый вывод решения:

```
► Эволюция вероятностей по времени (матрица-экспоненты):
t=0.00: P0=1.000000, P1=0.000000, P2=0.000000, sum=1.000000
t=1.01: P0=0.667468, P1=0.222196, P2=0.110336, sum=1.000000
t=2.51: P0=0.666667, P1=0.222222, P2=0.111111, sum=1.000000
t=5.03: P0=0.666667, P1=0.222222, P2=0.111111, sum=1.000000
t=7.54: P0=0.666667, P1=0.222222, P2=0.111111, sum=1.000000
t=10.00: P0=0.666667, P1=0.222222, P2=0.111111, sum=1.000000
► Эволюция вероятностей (метод Рунге–Кутта):
t=0.00: P0=1.000000, P1=0.000000, P2=0.000000, sum=1.000000
t=1.01: P0=0.667490, P1=0.222170, P2=0.110340, sum=1.000000
t=2.51: P0=0.666668, P1=0.222136, P2=0.111197, sum=1.000000
t=5.03: P0=0.666667, P1=0.222218, P2=0.111116, sum=1.000000
t=7.54: P0=0.666667, P1=0.222314, P2=0.111019, sum=1.000000
t=10.00: P0=0.666667, P1=0.222155, P2=0.111179, sum=1.000000
```

► Стационарные вероятности (равновесные):

P0=0.666667, P1=0.222222, P2=0.111111, sum=1.000000

Как видно, оба метода решения системы дифференциальных уравнений дают почти совпадающие результаты, при этом уже при значении времени t=10.00 вероятности практически достигают стационарных значений. Это подтверждается также графиками, представленными на рисунке 37. На графике кривые {bold p}(t), полученные двумя методами практически не различимы и при времени $t>2$ сливаются с прямыми стационарных вероятностей.

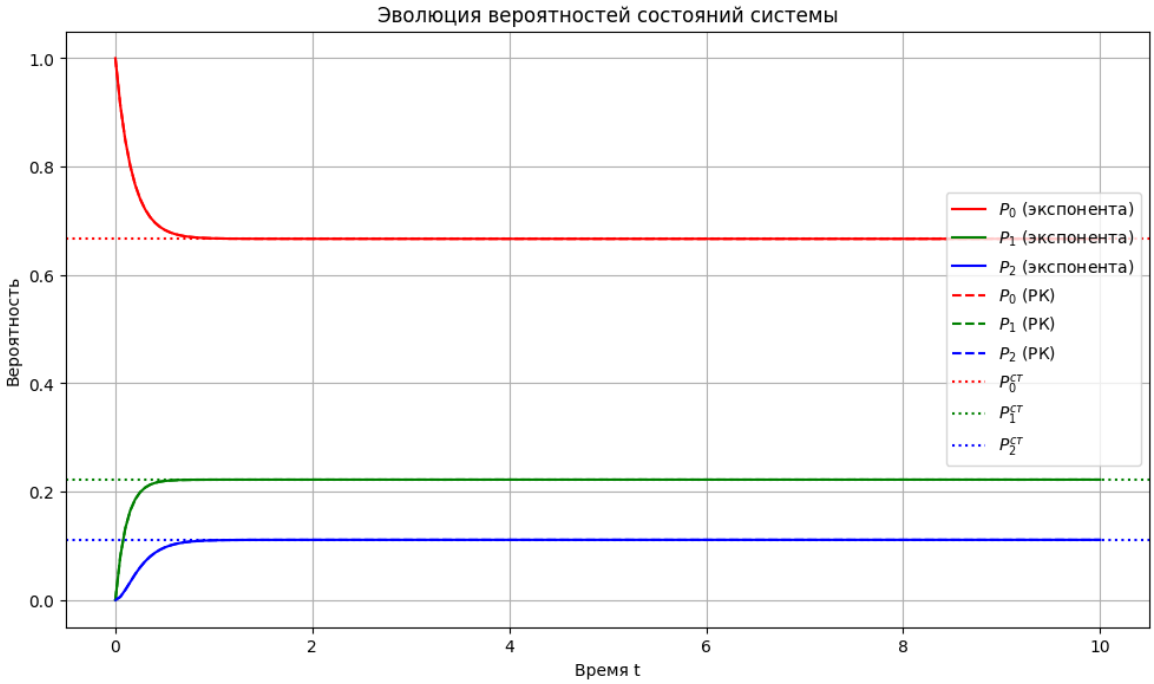

Рисунок 37 - Графики решений уравнений Колмогорова к задаче трехканальной СМО

6.2. Имитационные модели немарковских СМО

Удобство стандартных моделей СМО, в которых предполагается пуассоновское распределение требований и экспоненциальное распределение для времени обслуживания состоит в том, что эти допущения позволяют получить простые теоретические формулы для расчета показателей эффективности СМО, примеры которых приведены в таблицах 15-17. Однако на практике эти предположения выполняются далеко не всегда, в результате чего простые аналитические формулы для оценки показателей эффективности СМО не выполняются, а реальные

показатели существенно отличаются от теоретических. Расчеты таких систем осуществляются методами имитационного моделирования. Для иллюстрации рассмотрим следующую простую систему, достаточно часто встречающуюся в теории надежности технических систем.

Задача. Имеются два устройства: одно находится в работе, другое — в резерве. Через некоторое (случайное) время эксплуатации работающее устройство выходит из строя и отправляется в ремонт. В этот момент оно заменяется на резервное. Время безотказной работы одного устройства и время ремонта — случайные величины с заданными законами распределения. Считается что число ремонтов устройства неограниченно и после ремонта устройство полностью восстанавливает свои эксплуатационные характеристики (старение отсутствует). Полный отказ системы происходит в случае, когда оба устройства выходят из строя.

В случае, если время безотказной работы одного устройства и время ремонта описываются экспоненциальными распределениями с интенсивностями λ и μ соответственно данная задача может быть решена аналитически.

1. Среднее число работающих компонентов

В такой системе возможны состояния:

- $S=2$: оба компонента исправны;
- $S=1$: один работает, второй в ремонте или ожидании ремонта;
- $S=0$: оба вышли из строя ("полный отказ системы").

Чтобы вычислить среднее число работающих компонентов (статистику $\overline{S}$), находят стационарные вероятности для каждого состояния (p_2 , p_1 , p_0), решая систему балансовых уравнений.

Стационарные вероятности состояний:

- $p_2=\frac{\mu}{2\lambda+\mu}$
- $p_1=\frac{2\lambda}{2\lambda+\mu\cdot\frac{\mu}{\lambda+\mu}}$

- $p_0 = \frac{2\lambda}{2\lambda + \mu \cdot \frac{\lambda}{\lambda+\mu}}$

Тогда среднее число работающих компонентов:

$$\overline{S} = 2\,p_2 + 1\,p_1 + 0\,p_0 = 2\,p_2 + p_1$$

Подставив значения:

$$\overline{S} = 2 \cdot \frac{\mu}{2\lambda + \mu + \frac{2\lambda\mu}{(2\lambda+\mu)(\lambda+\mu)}}$$

Можно упростить до:

$$\overline{S} = \frac{2\mu(\lambda+\mu) + 2\lambda\mu}{(2\lambda+\mu)(\lambda+\mu)}$$

2. Среднее время до полного отказа системы (MTTF, Mean Time To Failure)

Когда оба компонента экспоненциально-надежны, а ремонты независимы, для системы "два компонента, cold standby", среднее время до первого полного отказа (то есть время работы, прежде чем оба компонента одновременно окажутся неисправны), вычисляется так :

$$MTTF = \frac{1}{\lambda}\left(1 + \frac{\mu}{\lambda}\right)$$

- Первый член ($1/\lambda$) — среднее время до первого отказа первого компонента;

- Второй член (μ/λ^2) — среднее время, когда второй компонент отказывает до окончания ремонта первого.

В случае не экспоненциальных распределений для времени безотказной работы одного устройства и времени ремонта, вышеописанные формулы не справедливы и показатели эффективности системы необходимо вычислять на основании имитационного моделирования.

Рассмотрим случай, когда распределение времени безотказной работы одного устройства описывается равномерным распределением в интервале $[0, T_{max\,fail}]$, а время ремонта T_{fix} фиксировано. Ниже приведен текст программы, в которой моделируется работа данной системы и результат сравнивается с моделью с экспоненциальным распределением времени безотказной работы и времени ремонта.

```
import random
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon
def simulate_ttf():
    S = 2
    SLast = 2
    Clock = 0.0
    TLast = 0.0
    Area = 0.0
    maxTfail=6
    Tfix=2.5
    NextFailure = maxTfail* random.random()
    NextRepair = math.inf
    while S > 0:
        if NextFailure < NextRepair:
            NextEvent = "Failure"
            Clock = NextFailure
```

```
            NextFailure = math.inf
        else:
            NextEvent = "Repair"
            Clock = NextRepair
            NextRepair = math.inf
        if NextEvent == "Failure":
            S -= 1
            if S == 1:
                NextFailure = Clock + maxTfail* random.random()
                NextRepair = Clock + Tfix
        else:
            S += 1
            if S == 1:
                NextFailure = Clock + maxTfail* random.random()
                NextRepair = Clock + Tfix
        Area += SLast * (Clock - TLast)
        TLast = Clock
        SLast = S
    return Clock, Area / Clock
def mm1_performance(lmbd, mu):
    p2 = mu / (2*lmbd + mu)
    p1 = (2*lmbd) / (2*lmbd + mu) * mu / (lmbd + mu)
    avg_working = 2*p2 + p1
```

```
    mttf = 1/lmbd * (1 + mu/lmbd)
    return avg_working, mttf
def mm1_ttf_pdf(x, lmbd, mu):
    # Теоретическая плотность времени до отказа двухкомп. standby системы:
    # f(t) = [lambda * (lambda + mu)] / (mu - lambda) * (exp(-lambda * t) - exp(-mu * t))
     # но это сложная формула; упрощенно — для СРАВНЕНИЯ по среднему используем просто exp-плотность
    mttf = 1/lmbd * (1 + mu/lmbd)
    # Экспоненциальная плотность (в среднем):
    return expon.pdf(x, scale=mttf)
def main(num_replications=1000):
    random.seed(1234)
    failure_times = []
    avg_working = []
    for _ in range(num_replications):
        ttf, avg_comp = simulate_ttf()
        failure_times.append(ttf)
        avg_working.append(avg_comp)
    failure_times = np.array(failure_times)
    avg_working = np.array(avg_working)
    mean_ttf = np.mean(failure_times)
    mean_working = np.mean(avg_working)
    var_ttf = np.var(failure_times, ddof=1)
    var_working = np.var(avg_working, ddof=1)
```

```
    # Параметры "экспоненциальной" модели
    lmbd = 1 / 3.5
    mu = 1 / 2.5
    mm1_avg_comp, mm1_mttf = mm1_performance(lmbd, mu)
    print(f"Симуляция ({num_replications} запусков):")
    print(f"  Среднее время до отказа: {mean_ttf:.4f}")
    print(f"  Дисперсия времени до отказа: {var_ttf:.4f}")
    print(f"  Среднее число работающих компонентов: {mean_working:.4f}")
    print(f"  Дисперсия числа работающих компонентов: {var_working:.4f}\n")
    print("Теоретическая экспоненциальная M/M/1-модель с теми же средними:")
    print(f"  Среднее время до отказа: {mm1_mttf:.4f}")
    print(f"  Среднее число работающих компонентов: {mm1_avg_comp:.4f}")
    # --- Рисуем гистограмму времени до отказа и теоретическую плотность экспоненты ---
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    # Гистограмма (частоты)
      n, bins, _ = plt.hist(failure_times, bins=30, density=True, color='skyblue', edgecolor='black',
label='Симуляция TTF')
    # Теоретическая экспонента для времени до отказа системы
    x = np.linspace(0, np.percentile(failure_times, 99.7), 200)
    plt.plot(x, mm1_ttf_pdf(x, lmbd, mu), 'r-', lw=2, label='Экспоненциальная M/M/1')
    plt.title('Время до отказа системы\n(сравнение с экспоненциальной моделью)')
    plt.xlabel('Время до отказа')
```

```
    plt.ylabel('Относительная частота/Плотность')
    plt.legend()
    # --- Гистограмма среднего числа работающих компонентов (frequency) ---
    plt.subplot(1, 2, 2)
     n2, bins2, _ = plt.hist(avg_working, bins=30, density=True, color='salmon', edgecolor='black', label='Симуляция TTF')
    # Теоретическая "плотность" (из-за дискретности -- просто вертикальная линия)
    plt.axvline(mm1_avg_comp, color='r', linestyle='--', lw=2, label='M/M/1 (аналитика)')
        plt.title('Среднее число работающих компонентов\n(сравнение с экспоненциальной моделью)')
    plt.xlabel('Среднее число компонентов')
    plt.ylabel('Относительная частота')
    plt.legend()
    plt.tight_layout()
    plt.show()
if __name__ == "__main__":
    main(10000)
```

Результат работы программы для параметров $T_{max\,fail}=6$, $T_{fix}=2.5$:

```
Симуляция (1000 запусков):
  Среднее время до отказа: 10.7607
  Дисперсия времени до отказа: 72.9181
  Среднее число работающих компонентов: 1.5670
  Дисперсия числа работающих компонентов: 0.0320
```

Теоретическая экспоненциальная M/M/1-модель с теми же средними:

Среднее время до отказа: 6.6000

Среднее число работающих компонентов: 1.0909

Как видно, стандартная экспоненциальная модель при тех же математических ожиданиях распределений времен долговечности и ремонта дает заниженные значения для среднего времени до отказа системы и среднего числа работающих компонентов. Это подтверждается также распределениями, приведенными на рисунке 38.

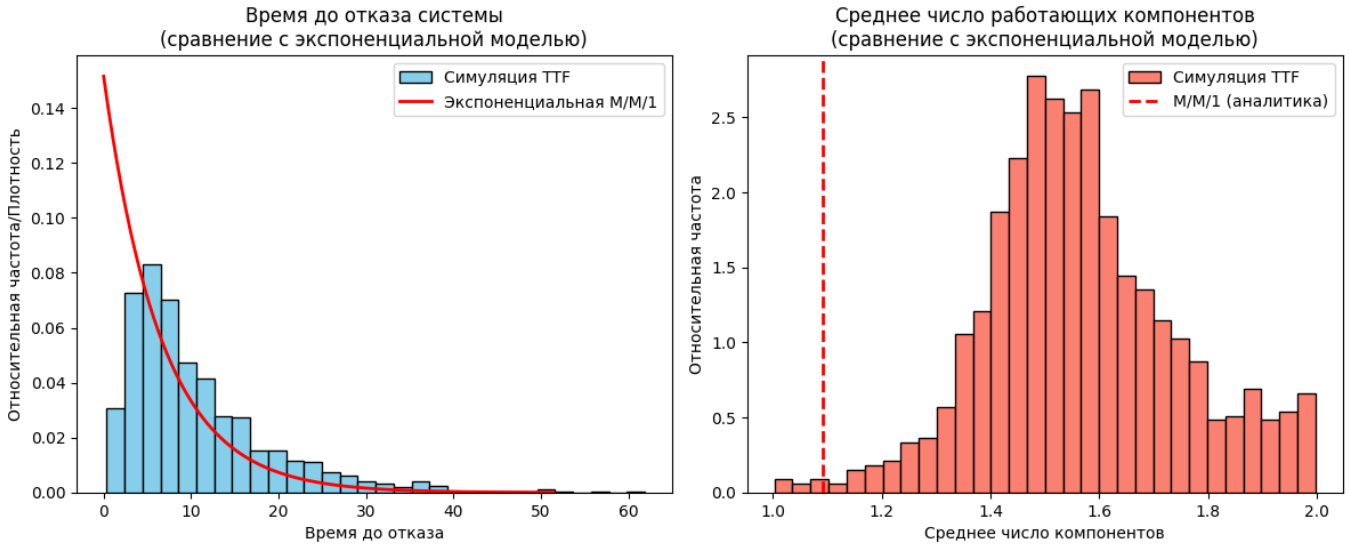

Рисунок 38 - Распределения среднего числа до отказов системы и среднего числа работающих элементов

Рассмотрим более сложный пример моделирования системы типа GI/GI/1 — с произвольными распределениями требований и времен обслуживания и зависящей от времени интенсивностью поступления заявок.

1. Постановка задачи

Требуется смоделировать работу одноканальной системы массового обслуживания (СМО) с неограниченной очередью в течение рабочего дня (с 8:00 до 20:00).

Особенности задачи:

- Входящий поток заявок генерируется с переменной (зависящей от времени суток) интенсивностью.

- Времена между поступлениями заявок имеют произвольное, не обязательно экспоненциальное распределение (например, эрланговское).

- Обслуживание поступившей заявки производится одним оператором. Время обслуживания равномерно распределено на заданном интервале [a, b] минут.

- Все поступающие заявки обязательно обслуживаются (отказов нет, очередь не ограничена по длине).

- Требуется провести имитационное моделирование, собрать статистику по длине очереди, времени ожидания, времени пребывания в системе и построить соответствующие графики.

2. Математическая формулировка

Входящие параметры

- Интервал моделирования: $T=[t_0,t_1]=[8:00,20:00]$ (в минутах с полуночи: 480–1200)

- Интенсивность входного потока заявок:

λ(t): кусочная функция, например:

λ(t) = 4 заявки/час, t ∈ [8:00, 12:00)

10 заявок/час, t ∈ [12:00, 18:00)

6 заявок/час, t ∈ [18:00, 20:00]

- Распределение времени между поступлениями (межприбытий):

Эрланговское:

τ ~ Erlang(k, θ(t)),

где $f_\tau(\tau)=\frac{1}{[\theta(t)]^k\cdot\frac{\tau^{k-1}}{(k-1)!}}\exp\left(-\frac{\tau}{\theta(t)}\right),\tau\geq 0$

- k — фиксировано (например, 2)

- Параметр $\theta(t)=60/(k\cdot\lambda(t))$.

- Время обслуживания:

U[a, b], равномерное распределение на интервале [a, b] минут.

Логика обслуживания

- Все заявки поступают в систему, формируют неограниченную FIFO-очередь.

- Один сервер (оператор) обслуживает заявки по мере готовности. Новая заявка начинает обслуживание как только оператор освобождается.

- Для каждой заявки фиксируются:

- Момент поступления (t_{arr})

- Момент начала обслуживания (t_{start})

- Момент окончания обслуживания (t_{end})

- Время ожидания ($t_{start}-t_{arr}$)

- Полное время пребывания в системе ($t_{end}-t_{arr}$)

Критерии эффективности (отчётные показатели):

- Среднее и максимальное время ожидания в очереди.

- Среднее время пребывания заявки в системе.

- Максимальная длина очереди.

- Графики: распределения поступлений, динамики очереди.

3. Описание (алгоритм) программы

1. Генерация поступлений:

- По кусочной интенсивности λ(t), для каждого поступления генерируется межприбытие по эрланговскому распределению с параметрами, соответствующими текущему времени.

- Процесс продолжается до конца рабочего дня.

2. Генерация обслуживания:

- Для каждой заявки генерируется длительность обслуживания по равномерному закону на [a, b].

3. Моделирование очереди:

- Для каждой заявки вычисляется возможное ожидание: если заявка приходит, когда оператор занят, она ждёт освобождения; иначе обслуживание начинается сразу.

- Фиксируются моменты начала и окончания обслуживания, а также время ожидания.

4. Ведение событий:

- Строится временной ряд изменения длины очереди для оценки динамики и максимальной нагрузки.

5. Визуализация и анализ:

- Строятся гистограммы и графики (например, распределения поступлений, длины очереди во времени).

- Выводится основная статистика по системе (средние, максимумы).

Ниже приведен текст программы, реализующей данную задачу:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Параметры поступления
start_time = 8 * 60    # минута дня (08:00)
end_time = 20 * 60     # минута дня (20:00)
def lambda_t(minute):
    if minute < 12 * 60:
        return 4   # утром
    elif minute < 18 * 60:
        return 10  # днём
    else:
        return 6   # вечером
def get_erlang_params(minute):
    k = 2
    lam = lambda_t(minute)
    mean_gap = 60.0 / lam
    scale = mean_gap / k
    return k, scale
# Параметры обслуживания (равномерное распределение)
service_time_low = 3    # в минутах (минимальное обслуживание)
service_time_high = 7   # в минутах (максимальное обслуживание)
# Генерация поступления
np.random.seed(0)
arrivals = []
current_minute = start_time
```

```
while current_minute < end_time:
    k, scale = get_erlang_params(current_minute)
    interarrival = np.random.gamma(shape=k, scale=scale)
    current_minute += interarrival
    if current_minute < end_time:
        arrivals.append(current_minute)
n = len(arrivals)
# Моделирование обслуживания (одна неограниченная очередь, один оператор)
service_times = np.random.uniform(service_time_low, service_time_high, size=n)
start_service_times = []
end_service_times = []
queue_wait_times = []
prev_end_time = arrivals[0]  # когда освободится оператор после первой заявки
for i in range(n):
    arrive = arrivals[i]
    if i == 0 or arrive >= prev_end_time:
        start_service = arrive           # Обслуживание сразу
    else:
        start_service = prev_end_time     # Ждём, когда оператор освободится
    wait_in_queue = start_service - arrive
    end_service = start_service + service_times[i]

    start_service_times.append(start_service)
```

```
    end_service_times.append(end_service)
    queue_wait_times.append(wait_in_queue)
    prev_end_time = end_service
# Посчитать показатели
mean_wait = np.mean(queue_wait_times)
max_wait = np.max(queue_wait_times)
mean_system_time = np.mean(np.array(end_service_times) - np.array(arrivals))
max_queue_length = 0
current_queue = 0
queue_lengths = []
# Для гистограммы занятости очереди — циркулируем по временам (дискретно)
events = []
for i, t in enumerate(arrivals):
    events.append( ('arr', t) )
for i, t in enumerate(end_service_times):
    events.append( ('dep', t) )
events.sort(key=lambda x: x[1])

queue_size = 0
queue_over_time = []
timeline = []
for event, t in events:
    if event=='arr':
```

```
        queue_size += 1
    else:
        queue_size -= 1
    timeline.append(t)
    queue_over_time.append(queue_size)


max_queue_length = max(queue_over_time)
# График поступлений
plt.figure(figsize=(10,2.5))
plt.hist(np.array(arrivals)/60, bins=np.arange(8,20,0.5), color='skyblue', rwidth=0.95)
plt.xlabel('Время (часы от 8:00)')
plt.ylabel('Число заявок за 30 мин')
plt.title('Входящие заявки по времени')
plt.show()
# График длины очереди во времени
plt.figure(figsize=(10,2.5))
plt.step(np.array(timeline)/60, queue_over_time, where='post')
plt.xlabel('Время (часы)')
plt.ylabel('Число заявок в очереди')
plt.title('Динамика очереди (столбец — конец события)')
plt.show()
# Итоговая статистика
print(f"Всего заявок: {n}")
```

```
    print(f"Среднее время ожидания в очереди: {mean_wait:.2f} минут")
    print(f"Максимальное ожидание: {max_wait:.2f} минут")
    print(f"Среднее время заявки в системе: {mean_system_time:.2f} минут")
    print(f"Максимальная длина очереди: {max_queue_length}")
    # Пример первых 10 заявок (форматированные времена)
    print("\nПервые 10 заявок:")
    for i in range(min(10, n)):
        arr_h, arr_m = int(arrivals[i]//60), int(arrivals[i]%60)
        start_h, start_m = int(start_service_times[i]//60), int(start_service_times[i]%60)
        end_h, end_m = int(end_service_times[i]//60), int(end_service_times[i]%60)
            print(f"Заявка {i+1:2d}: поступила {arr_h:02d}:{arr_m:02d}, обслуживание с {start_h:02d}:{start_m:02d}, окончено {end_h:02d}:{end_m:02d}, ожидание: {queue_wait_times[i]:.2f} мин.")
```

Вывод результатов выполнения:

```
Всего заявок: 83
Среднее время ожидания в очереди: 8.54 минут
Максимальное ожидание: 30.76 минут
Среднее время заявки в системе: 13.55 минут
Максимальная длина очереди: 8
Первые 10 заявок:
Заявка  1: поступила 08:38, обслуживание с 08:38, окончено 08:41, ожидание: 0.00 мин.
Заявка  2: поступила 08:55, обслуживание с 08:55, окончено 09:00, ожидание: 0.00 мин.
Заявка  3: поступила 09:36, обслуживание с 09:36, окончено 09:40, ожидание: 0.00 мин.
```

Заявка 4: поступила 09:41, обслуживание с 09:41, окончено 09:47, ожидание: 0.00 мин.

Заявка 5: поступила 09:52, обслуживание с 09:52, окончено 09:57, ожидание: 0.00 мин.

Заявка 6: поступила 10:09, обслуживание с 10:09, окончено 10:13, ожидание: 0.00 мин.

Заявка 7: поступила 10:31, обслуживание с 10:31, окончено 10:34, ожидание: 0.00 мин.

Заявка 8: поступила 10:44, обслуживание с 10:44, окончено 10:48, ожидание: 0.00 мин.

Заявка 9: поступила 11:02, обслуживание с 11:02, окончено 11:07, ожидание: 0.00 мин.

Заявка 10: поступила 11:18, обслуживание с 11:18, окончено 11:22, ожидание: 0.00 мин.

На рисунках 39, 40 приведены полученные в результате моделирования гистограммы распределения заявок и длины очереди по времени дня.

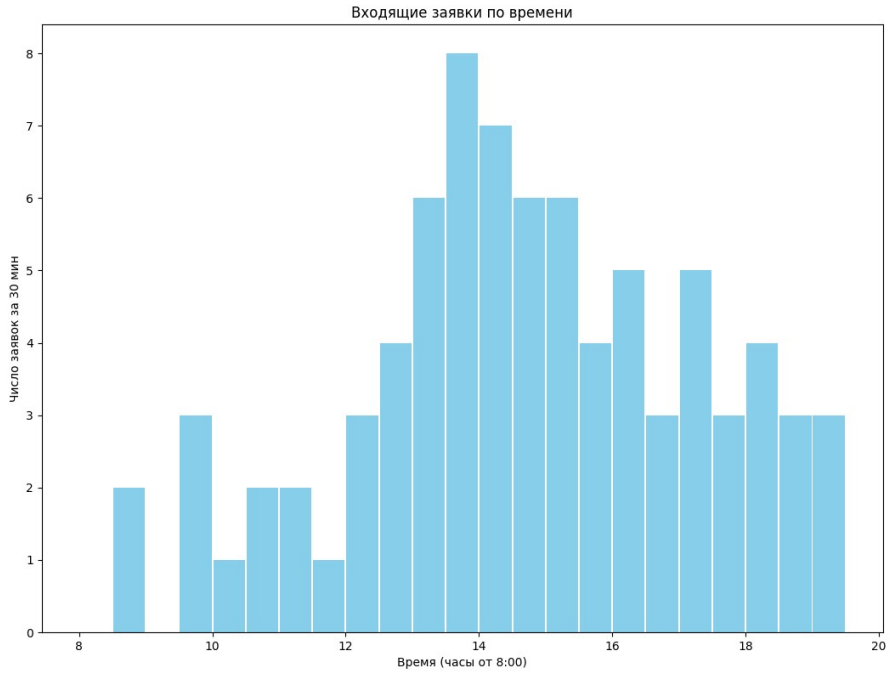

Рисунок 39 - Гистограмма распределения заявок по времени дня

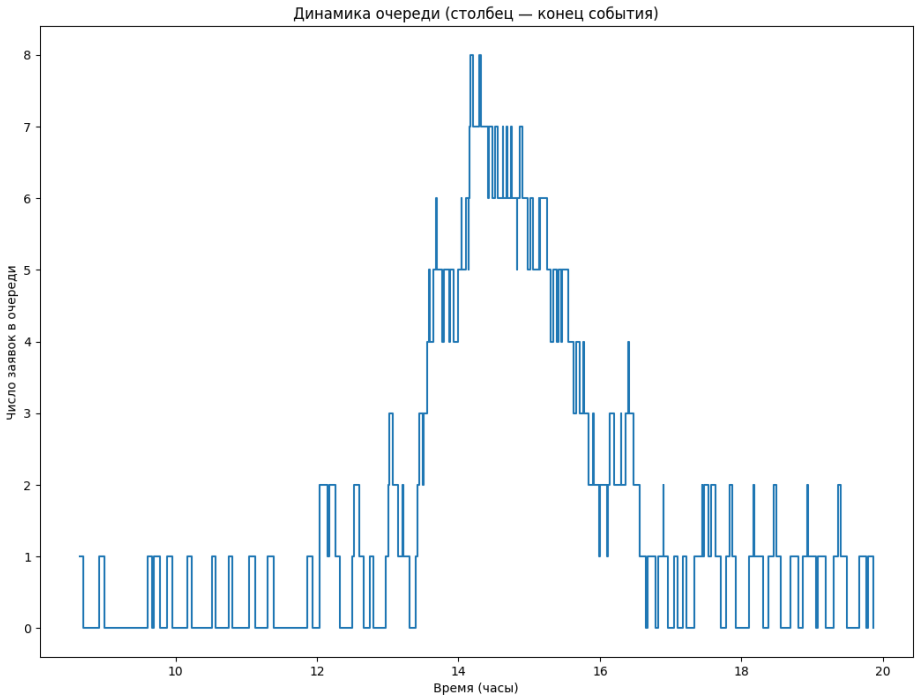

Рисунок 40 - Распределение длины очереди по времени дня

Комментарии к полученным результатам

- Гистограмма поступлений чётко показывает неравномерность потока: пики и спады интенсивности в соответствующие периоды дня.

- График длины очереди во времени визуализирует динамику загрузки системы: когда интенсивность большая (днём), очередь может существенно увеличиваться, если среднее время обслуживания близко к среднему межприбытия.

- Средние показатели (время ожидания, пребывания в системе, длина очереди) позволяют оценить эффективность работы оператора и удобство для клиентов:

- Если среднее ожидание велико — значит, серверу не хватает мощности в часы пиковой нагрузки (можно рассматривать увеличение штата/скорость обслуживания).

- Если ожидание и очередь невелики, а оператор часто простаивает — возможно, ресурсов избыточно и есть потенциал оптимизации расходов.

- Максимальное ожидание и длина очереди важны для оценки экстремальных случаев и риска возникновения "заторов" или утомительных ожиданий клиентов.

Итоговая трактовка

Имитационное моделирование позволяет количественно и наглядно оценить поведение реальной очереди при сложных потоках заявок (произвольные межприбытия, нестационарная интенсивность, любые законы обслуживания). Такая модель помогает:

- Определить «узкие места»;

- Оценить, как изменится нагрузка, если, например, увеличить время обслуживания или интенсивность заявок;

- Принять технические/организационные решения (добавить сотрудников, изменить расписание и т.д.);

- Оценить удовлетворённость клиентов по среднему времени ожидания.

При необходимости можно расширять модель и статистику, вводить нескольких операторов, ограничивать очередь, учитывать приоритеты, задавать неравномерные распределения обслуживания и т.д.

Заключение

В настоящем учебном пособии рассмотрены ключевые аспекты статистического анализа и имитационного моделирования с использованием языка программирования Python. Основное внимание уделено методам обработки и интерпретации данных, а также их применению в различных областях, таких как инженерия, медицина, биология и информационные технологии.

Пособие начинается с введения в Python, включая основы синтаксиса, работу с данными и объектно-ориентированное программирование. Далее представлены математические основы теории вероятностей и статистики, необходимые для понимания последующих глав. Особое внимание уделено описанию распределений случайных величин, их параметров и методов анализа.

В главах, посвященных статистическому анализу, подробно рассмотрены методы оценки параметров распределений, проверки гипотез, дисперсионного и регрессионного анализа. Отдельно обсуждаются вопросы планирования экспериментов, что является важным этапом в исследованиях и практических приложениях.

Заключительные главы охватывают методы оптимизации, нелинейной регрессии, нейронных сетей и имитационного моделирования. Приведены примеры использования специализированных библиотек Python, таких как NumPy, SciPy, Pandas и Matplotlib, что позволяет читателю не только освоить теоретические основы, но и применить их на практике.

При построении изложения особое внимание уделяется постепенному переходу от базовых математических понятий (множества, меры, функции распределения, одномерные и многомерные случайные величины) к алгоритмам практического анализа выборок, построения доверительных интервалов, проверки статистических гипотез и выявления связей и влияний в сложных системах. Большое количество иллюстрированных примеров на Python обеспечивает максимальную наглядность излагаемого материала, а присутствие кода и легких для адаптации алгоритмов делает пособие не только теоретической, но и практической опорой для самостоятельной работы студентов и исследователей.

Методические аспекты учебника построены с учетом современных образовательных требований: каждый ключевой раздел сопровождает описание алгоритмов вычислений, демонстрация работы с инструментами Python (NumPy, Pandas, SciPy, StatsModels, Matplotlib и др.), что позволяет применять статистические, регрессионные, оптимизационные и симуляционные

методы во множестве профессиональных областей — от инженерии, физики и биомедицины до экономики, социальных и информационных технологий.

Для дальнейшего изучения и проектной работы можно было бы рекомендовать следующие направления:

- Машинное обучение на основе Python (обработка больших данных, статистическое и имитационное моделирование для Data Science).
- Имитация сложных производственных, экономических, биомедицинских процессов на Python с использованием open source библиотек.
- Автоматизация и визуализация анализа данных при помощи интерактивных инструментов (Jupyter, Google Colab).
- Методы многофакторного планирования эксперимента и оптимизации параметров систем в инженерии и науке.

Учебное пособие предназначено для студентов и аспирантов, изучающих информационные системы и технологии, а также для специалистов, занимающихся математическим моделированием и анализом данных.

Список использованных источников

1. Ширяев, А.Н. Вероятность - 1 / А.Н. Ширяев.- М.: МЦНМО, 2004.- 520 с.
2. Ширяев, А.Н. Вероятность - 2 / А.Н. Ширяев.- М.: МЦНМО, 2004.- 408 с.
3. Кремер, Н.Ш. Теория вероятностей и математическая статистика / Н.Ш. Кремер.- М.: ЮНИТИ-ДАНА, 2004.- 573 с.
4. R: A Language and Environment for Statistical Computing. Reference Index [Электронный ресурс].- Режим доступа: https://lib.stat.cmu.edu/R/CRAN/doc/manuals/fullrefman.pdf%20 (дата обращения: 24.07.2024)
5. pandas [Электронный ресурс].- Режим доступа: https://pandas.pydata.org/ (дата обращения: 24.07.2024)
6. NumPy [Электронный ресурс].- Режим доступа: https://numpy.org/ (дата обращения: 24.07.2024)
7. SciPy [Электронный ресурс].- Режим доступа: https://scipy.org/ (дата обращения: 24.07.2024)
8. statsmodels [Электронный ресурс].- Режим доступа: https://www.statsmodels.org/stable/index.html (дата обращения: 24.07.2024)
9. Matplotlib: Visualization with Python [Электронный ресурс].- Режим доступа: https://matplotlib.org/ (дата обращения: 20/07/2024)
10. seaborn: statistical data visualization [Электронный ресурс].- Режим доступа: https://seaborn.pydata.org/ (дата обращения: 24.07.2024)
11. AnyLogic: имитационное моделирование для бизнеса [Электронный ресурс].- Режим доступа: https://www.anylogic.ru/ (дата обращения: 24.07.2024)
12. NetLogo [Электронный ресурс].- Режим доступа: https://ccl.northwestern.edu/netlogo/ (дата обращения: 24.07.2024)
13. PySCeS [Электронный ресурс].- Режим доступа: https://pysces.sourceforge.net/ (дата обращения: 24.07.2024)
14. Simul8 decisions [Электронный ресурс].- Режим доступа: https://www.simul8.com/ (дата обращения: 24.07.2024)
15. Кудрявцев, Е. М. GPSS World. Основы имитационного моделирования различных систем. / Е. М. Кудрявцев.- М.: ДМК Пресс, 2004.- 320 с.
16. SimPy [Электронный ресурс].- Режим доступа: https://simpy.readthedocs.io/en/latest/ (дата обращения: 24.07.2024)
17. Salabim [Электронный ресурс].- Режим доступа: https://www.salabim.org/ (дата обращения: 24.07.2024)
18. PySD [Электронный ресурс].- Режим доступа: https://pysd.readthedocs.io/en/master/ (дата обращения: 24.07.2024)
19. BPTK_Py [Электронный ресурс].- Режим доступа: https://pypi.org/project/BPTK-Py/ (дата обращения: 24.07.2024)
20. Tobias, S.L. Open-source discrete-event simulation software for applications in production and logistics / S.L. Tobias, R.M. Müller, A.O. von Guericke // Procedia Computer Science.- 2021.- V. 180 . - P. 978–987
21. ARENA [Электронный ресурс].- Режим доступа: https://arena.run/ (дата обращения: 24.07.2024)
22. Колмогоров, А. Н. Основные понятия теории вероятностей / А. Н. Колмогоров.- М.: Наука, 1974.- 120 с.
23. Чернова, Н. И. Теория вероятностей / Н. И. Чернова.- Новосибирск: Новосиб. гос. ун-т, 2007.- 160 с.
24. Shapiro, S.S. An appriximate analysis of variance test fo normality / S.S. Shapiro, R.S. Francia // J. Amer. Statist. Assoc..- 1972.- V. 337 . - P.215-216

25. Kolmogorov, A. N. Confidence liшits for an unknown distribution function / A. N. Kolmogorov // AMS.- 1941.- V.12 . - P. 461-463
26. Смирнов, Н. В. Оценка расхождения между эмпирическими кривыми распределений / Н. В. Смирнов // Бюллетень МГУ. Сер. А.- 1939.- Вып. 2 . - С. 13-14
27. Wilcoxon, F. Individual comparisons by ranking methods / F. Wilcoxon // Biometrics.- 1945.- V. 1 . - P. 80-83
28. Mann, H. B. On a test of whether one of two random variabes is stochastically larger than the other / H.B. Mann , D. R. Whitney // AMS.- 1947.- V. 18 . - P. 50-60
29. Шеффе, Г. Дисперсионный анализ / Г. Шеффе.- М.: Наука, 1980.- 512 с
30. Ермаков, С.М. Математическая теория оптимального эксперимента / С.М. Ермаков, А.А. Жиглявский.- М.: Наука, 1987.- 320 с
31. Асатурян, В.И. Теория планирования эксперимента / В.И. Асатурян.- М.: Радио и связь, 1983.- 248 с
32. Ермаков, С.М. Математическая теория планирования эксперимента / С.М. Ермаков, В.З. Бродский, А.А. Жиглявский и др..- М.: Наука, 1983.- 392 с
33. Пантелеев, А.В. Методы оптимизации в примерах и задачах / А. В. Пантелеев, Т. А. Летова.- М.: Высшая школа, 2018.- 544 с.
34. Гасников, А.В. Современные численные методы оптимизации. Метод универсального градиентного спуска: учебное пособие / А. В. Гасников.- М.: МФТИ, 2021.- 160 с.
35. Черноруцкий, И. Г. Методы оптимизации в теории управления / И. Г. Черноруцкий.- СПб.: Питер, 2022.- 256 с.
36. Nelder, J.A. A Simplex Method for Function Minimization / J.A. Nelder, R.Mead // Comput. J.- 1965.- V.7 . - P. 308-313
37. Seber, G. A. F. Nonlinear regression / G. A. F. Seber, C. J. Wild .- Hoboken, NJ: Wiley-Interscience, 2003.- 768 p.
38. Draper N. R. Applied regression analysis / N.R. Draper, H. Smith.- New York: Wiley, 1998.- 713 p.
39. Bates D. M. Nonlinear regression analysis and its applications / D. M. Bates, D.G. Watts.- New York: Wiley, 1988.- 365 p.
40. Hastie, T. The elements of statistical learning: data mining, inference, and prediction / T. Hastie, R. Tibshirani, J. Friedman .- New York: Springer, 2009.- 745 p.
41. Grus J. Data Science from Scratch / .- O'Reilly, 2015.- 330 p.
42. Постолит А.В. Основы искусственного интеллекта в примерах на Python. Самоучитель. / .- СПб.: БХВ-Петербург, 2021.- 448 с.
43. Гафаров Ф.М., Галимянов А.Ф. Искусственные нейронные сети и приложения: учеб. пособие / .- Казань: Изд-во Казан. ун-та, 2018.- 121 с.
44. Kanta A.-F., Montavon G., Coddet C. Predicting Spray Processing Parameters from Required Coating Structural Attributes by Artificial Intelligence / // Advanced Engineering Materials.- 2006.- V.8. no. 7. - P.628-635
45. Кельтон В., Лоу А. Имитационное моделирование. Классика CS / .- СПб: Питер, 2004.- 847 с
46. Марголис Н.Ю. Имитационное моделирование : учеб. пособие / .- Томск : Издательский Дом Томского гос., 2015.- 130 с.