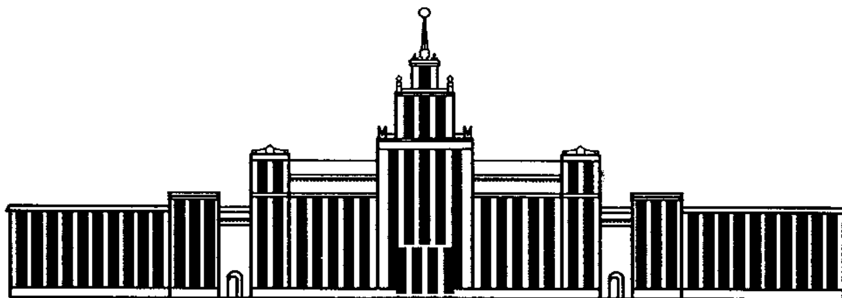
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ



ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

519.6(07)
С544

А.Н. Соболев, А.Г. Воронцов

КОМПЬЮТЕРНАЯ ФИЗИКА

Учебное пособие

Челябинск
2016

Министерство образования и науки Российской Федерации
Южно-Уральский государственный университет
Кафедра общей и теоретической физики

519.6(07)
С544

А.Н. Соболев, А.Г. Воронцов

КОМПЬЮТЕРНАЯ ФИЗИКА

Учебное пособие

Челябинск
Издательский центр ЮУрГУ
2016

УДК 519.6(075.8) + [51-72:53](075.8)
С544

Одобрено
учебно-методической комиссией
физического факультета

Рецензенты:
А.Е. Майер, Б.Р. Гельчинский

Соболев, А.Н.
С544 Компьютерная физика: учебное пособие / А.Н. Соболев, А.Г. Воронцов. – Челябинск: Издательский центр ЮУрГУ, 2016. – 119 с.

Данное пособие объединяет материал, находящийся на стыке трех наук: физики, математики и информатики. Первая часть пособия посвящена описанию конструкций языка программирования Python. Во второй части изложены особенности использования языка Python для решения широкого класса математических задач. В третьей части рассматривается методика использования компьютера при решении физических задач. В каждом разделе имеются примеры и задания, выполнение которых позволит быстрее освоить материал. При выборе примеров основное внимание уделено задачам, наиболее часто встречающимся в практике ученого-физика.

Пособие предназначено для бакалавров направления «Прикладные математика и физика»; кроме этого, оно будет полезно всем студентам, интересующимся применением компьютеров для решения физических задач.

УДК 519.6(075.8) + [51-72:53](075.8)

© Издательский центр ЮУрГУ, 2016

Оглавление

Введение **5**

1. Язык программирования Python

1.1. Общая информация 8
1.2. Установка и настройка 9
1.3. Основные элементы языка 14
1.4. Объекты и классы 22
1.5. Элементы функционального программирования 23
1.6. Модули 28

2. Научные вычисления в Python

2.1. Numpy. Общая информация 31
2.2. Массивы Numpy 32
2.3. Дополнительные модули NumPy 39
2.4. Пакет Scipy 44
2.5. Библиотека Matplotlib. Общая информация 56
2.6. Интерфейс Pyplot 57
2.7. Объекты Matplotlib 60
2.8. Анимация в Matplotlib 62

3. Решение физических задач

3.1. Применение компьютера для решения физических задач . . 65
3.2. Задачи на движение отдельных тел 71
3.3. Задачи на движение сплошных сред и нахождение полей . . 79

Заключение **92**

А. Справочник команд Python

А.1. Математические операции 94
А.2. Логические операции 94
А.3. Логические условия 95
А.4. Доступ к элементам последовательностей 95
А.5. Функции для работы с последовательностями 95
А.6. Форматирование строк 96

B. Справочник команд Numpy
B.1. Создание массивов из имеющихся данных 98
B.2. Создание массивов определённого вида 99
B.3. Создание сеток 101
B.4. Трансформации массива без изменения элементов 103
B.5. Слияние и разделение массивов 104
B.6. Алгебраические функции для массивов 105
B.7. Тригонометрические функции для массивов 106
B.8. Бинарные функции для массивов 107
B.9. Другие функции для массивов 108
B.10. Сортировка, поиск, подсчет 109
B.11. Статистика 111
B.12. Дискретное преобразование Фурье (numpy.fft) 112
B.13. Линейная алгебра (numpy.linalg) 113
B.14. Случайные величины (numpy.random) 115
B.15. Полиномы (numpy.polynomial) 117

C. Краткий справочник Matplotlib
C.1. Некоторые аргументы `pyplot.plot` 119
C.2. Некоторые функции интерфейса `pyplot` 119

Введение

Данное учебное пособие разработано в соответствии ФГОС ВО по направлению подготовки 03.03.01 «Прикладные математика и физика» (уровень бакалавра). Целью представленного пособия является подготовка студента к решению ряда профессиональных задач, в соответствии с научно-исследовательским направлением деятельности:

- участие в проведении теоретических исследований, построении физических, математических и компьютерных моделей изучаемых процессов и явлений, в проведении аналитических исследований в предметной области по профилю специализации;
- участие в обобщении полученных данных, формировании выводов, в подготовке научных и аналитических отчетов, публикаций и презентаций результатов научных и аналитических исследований;
- участие в разработке новых алгоритмов и компьютерных программ для научно-исследовательских и прикладных целей.

Подготовка к студента к решению указанного круга задач происходит через формирование ряда общепрофессиональных и профессиональных компетенций, соответствующих научно-исследовательской деятельности:

- способностью применять теорию и методы математики для построения качественных и количественных моделей объектов и процессов в естественнонаучной сфере деятельности (ОПК-2);
- способностью применять полученные знания для анализа систем, процессов и методов (ОПК-4);
- способностью представлять результаты собственной деятельности с использованием современных средств, ориентируясь на потребности аудитории, в том числе в форме отчетов, презентаций, докладов (ОПК-6).
- способностью планировать и проводить научные эксперименты (в избранной предметной области) и (или) теоретические (аналитические и имитационные) исследования (ПК-1);
- способностью анализировать полученные в ходе научно-исследовательской работы данные и делать научные выводы (заключения) (ПК-2);
- способностью критически оценивать применимость применяемых методик и методов (ПК-4);

Данное пособие объединяет материал, находящийся на стыке трех наук: физики, математики и информатики. Физика описывает практическую ситуацию, математика дает формальный способ ее исследования, а информатика является инструментом, который позволяет получить ответ. В данном пособии рассмотрен наиболее эффективный, с точки зрения ученого–физика, способ использования компьютера.

Первая часть пособия посвящена описанию конструкций языка программирования Python. Материал снабжен многочисленными примерами и заданиями, выполнение которых позволит быстрее освоить использование данного языка.

Во второй части рассматриваются особенности использования языка Python в научных исследованиях. Приведено описание трех наиболее востребованных модулей, предназначенных для выполнения операций с массивами и матрицами (Numpy), решения задач математического анализа (Scipy), и визуализации данных (Matplotlib). В каждом разделе имеются примеры и задания. При выборе примеров основное внимание уделено задачам, наиболее часто встречающимися в практике ученого–физика: построение графиков и визуализация хода работы программы, работа с большими объемами данных, использование готовых алгоритмических решений для научных вычислений.

В третьей части рассматривается методика использования компьютера при решении физических задач. Даны общие методические рекомендации по всем этапам решения задачи от анализа физической проблемы и математической формулировки до составления алгоритма и тестирования. После краткого теоретического введения, снабженного ссылками на дополнительные литературные источники, приводятся алгоритмы и фрагменты кода программ, позволяющих осуществить вычисления по указанным алгоритмам. В конце разделов даны задания для самостоятельного численного исследования свойств простых динамических систем. Формулировка заданий позволяет уделить внимание не только анализу физических проблем, но и совершенствованию алгоритмов и оптимизации вычислений.

В результате работы с данным пособием студенты должны познакомится с методиками применения ЭВМ при проведении численного анализа задач, встречающихся в практике физика-теоретика, научится формализовывать физические задачи и сводить их к формальным математическим, составлять алгоритмы для решения математических задач и овладеть основами использования языка программирования Python для решения типичных задач, по реализации алгоритмов, манипуляции данными представленными в разных форматах, графическому представлению результатов моделирвания, и визуализации физических процессов.

Преподавателям при работе с пособием рекомендуется ориентироваться на выполнение студентами практических заданий. Теоретические части даны в первую очередь для базового ознакомления с методами, реализованными в известных алгоритмах, однако их достаточно для работы с материалом пособия. Разный уровень заданий позволяет индивидуально подобрать их в соответствии с уровнем подготовки студента.

Данное пособие относится к курсу «Компьютерная физика» в рамках базовой части основной образовательной программы по направлению 03.03.01 «Прикладные математика и физика». Перед изучением данного курса студенты должны освоить общий курс физики, элементы математического анализа и линейной алгебры, основы вычислительной математики.

1. Язык программирования Python

1.1. Общая информация

Python (произносится «па́йтон») — созданный в 1991 году интерпретируемый язык высокого уровня со строгой динамической типизацией. Создатель языка Гвидо ван Россум так описывает его:

> **Python** — легко изучаемый, но в то же время мощный язык программирования. Данные в нем огранизуются в эффективные высокоуровневые структуры; реализованный подход к объектно-ориентированному программированию прост, но эффективен. Интерпретируемая природа языка в сочетании с простым синтаксисом и динамической типизацией делают язык привлекательным для быстрой разработки как скриптов, так и полноценных приложений во многих сферах деятельности на большинстве платформ.

Перечислим особенности языка Python.

- *Интерпретируемость*
  Исходный код программ на Python не преобразовывается в машинный код для непостредственного его выполнения центральным процессором, а выполняется специальной программой — интерпретатором. Благодаря тому, что интерпретатор языка портирован под разные архитектуры, написанная на Python программа может быть без изменений запущена на разных платформах, начиная от Windows, и заканчивая Android. Однако, за это приходится расплачиваться *медленностью* выполнения программы по сравнению с компилируемыми языками, такими, как C или Fortran.
- *Высокий уровень языка*
  Программирование на высоком уровне абстракции позволяет сконцентрироваться на решении задачи и не задумываться о низкоуровневых процессах, происходящих в системе (например, выделении и освобождении памяти), что приводит к написанию простого и читаемого кода.
- *Читаемость кода*
  Синтаксис языка Python очень прост и похож на синтаксис английского языка. Вложенность участков кода определяется *отступами*.
- *Поддержка различных парадигм*

Python поддерживает различные стили программирования: императивный (скриптовый; программа представляет собой набор инструкций, изменяющих состояние данных), объектно-ориентированный (основанный на концепции объектов — структур, содержащих данные (атрибуты объекта) и код, определяющий его поведение (методы объекта)) и функциональный (основанный на последовательном вычислении результатов функций). Сам Python — объектно-ориентированный язык в том смысле, что каждый элемент языка представляет собой объект.

- *Большое сообщество*
Благодаря тому, что Python — свободно распространяемый продукт, вокруг него образовалось большое сообщество, включающее такие компании, как Google, Apple, IBM, CERN, Яндекс и др. Это привело к тому, что к настояшему моменту для Python написано очень большое количество модулей (библиотек), покрывающих все сферы использования компьютеров.

1.2. Установка и настройка

Установка Python

Установка Python на компьютер обычно включает в себя установку интерпретатора языка и дополнительных модулей, если они необходимы для работы. В настоящее время существуют и развиваются две версии языка: Python 2 и Python 3. Синтаксис Python 3 обратно не совместим со второй версией языка[1]. В рамках этого курса мы будем использовать Python 2. В следующих параграфах описана установка и настройка интерпретатора Python 2 на различных платформах.

Windows

Стандартный установщик интерпретатора Python можно скачать с официального сайта Фонда развития Python:

`www.python.org/downloads/`.

Хотя в стандартную комплектацию уже включено большое количество модулей, порой их бывает недостаточно для полноценной работы. Так, в научной среде часто используются модули Numpy (для матричных вычислений), Scipy (для сложных научных расчетов) и Matplotlib (для построения графиков). Эти модули написаны на C для ускорения

[1]Причины этого можно узнать в официальном сообщении Гвидо ван Россума: `www.artima.com/weblogs/viewpost.jsp?thread=208549`.

их работы, и поэтому требуют компиляции для их использования. Для большинства модулей на страницах в Интернете предлагаются уже скомпилированные версии, однако скачивание и установка каждого необходимого модуля неудобно и ведет к беспорядку в операционной системе. Вследствие этого некоторые компании предлагают свои «сборки» интерпретатора Python, дополненные большим количеством дополнительных модулей и программой, позволяющей их централизованную установку и удаление. Перечислим некоторые из таких «сборок»:

- Enthought Canopy — сборка из более 100 (в бесплатной версии) либо 300 (в полной версии) модулей для научных вычислений, анализа и визуализации данных. Полная версия бесплатна для академического использования.
  Сайт — `www.enthought.com`.
- Python(x,y) — бесплатная сборка Python для использования в научной среде.
  Сайт — `code.google.com/p/pythonxy/`
- Anaconda от Continuum Analytics — свободно распространяемая сборка из 295 наиболее используемых модулей в математике, естественных науках и анализе данных. Также существует в минимальной конфигурации Miniconda, включающей в себя только интерпретатор и программу `conda` для централизованного управления пакетами.
  Сайт — `continuum.io/downloads`

В рамках этого курса мы будем использовать Miniconda. Процесс установки состоит в следующем:

1. Скачайте с сайта

   `http://conda.pydata.org/miniconda.html`

   последнюю версию Miniconda, основанную на Python 2.7 (32- или 64-битную в зависимости от установленной версии Windows).
2. Запустите скачанный файл установщика.
3. Примите условия лицензионного соглашения.
4. Выберите вариант установки: для конкретного пользователя либо для всех пользователей компьютера (этот вариант предпочтительней в случае, если в имени пользователя есть русские буквы).
5. Выберите путь для установки Miniconda или оставьте предложенный по умолчанию.
6. Поставьте две галочки: для добавления пути Miniconda в переменную окружения PATH и для регистрации ее как интерпретатора Python по умолчанию в системе.

После этого у вас в системе будет установлен интерпретатор Python. Для проверки правильности установки нажмите Win+R (Win — кнопка

с логотипом Microsoft Windows) и в появившемся окне наберите python. При удачном выполнении команды должно появиться окно терминала с приглашением к вводу >>> (номер версии может быть другим):

```
Python 2.7.9 |Continuum Analytics, Inc.| (default, Dec 18 2014, 16:57:52)
[MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>>
```

Linux

В современных дистрибутивах Linux интерпретатор Python установлен по умолчанию. Для того, чтобы проверить это, необходимо набрать python в окне терминала. После нажатия Enter должно появиться приглашение к вводу команд >>>. Если же вместо этого вы видите что-то, похожее на

```
bash: python: Command not found,
```

это означает, что интерпретатор Python на компьютер не установлен. Для его установки необходимо использовать менеджер пакетов вашего дистрибутива: yum (Fedora до 21-й версии), dnf (Fedora 22 и старше), apt-get (Debian, Ubuntu) и т.д. Команда

```
sudo yum install python
```

установит интерпретатор Python на компьютер под управлением Fedora Linux. Заметьте, что пользователь должен находиться в группе wheel — группе пользователей, которым разрешено использовать команду sudo. После установки интерпретатора Python на компьютер проверьте правильность установки указанным выше способом.

Начало работы с Python

Для начала работы с Python запустим интерпретатор. Он запускается в т.н. *«интерактивном режиме»* (REPL; Read-Eval-Print Loop) — режиме, при котором любая команда выполняется сразу после введения. Интерпретатор в этом режиме можно использовать как продвинутый калькулятор:

```
>>> 2 * 2
4
```

Результат расчета появляется сразу после нажатия клавиши Enter. Арифметика длинных чисел поддерживается прозрачно для пользователя:

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

`**` в предыдущем листинге обозначает операцию возведения в степень. Математические операторы, используемые в Python, приведены в приложении А.1. Обратите внимание на разницу между `/` (оператор деления) и `//` (оператор целочисленного деления).

Для хранения данных в процессе вычисления могут использоваться переменные различных типов (подробнее см. гл. 1.3)

Пример 1.1. Автомобиль проехал 210 км за 3 часа 30 минут. Чему равна его средняя скорость?

Для удобства введем переменные: s — пройденное расстояние, t — затраченное время (3 часа 30 минут = 3,5 часа). Тогда средняя скорость будет равна s/t:

```
>>> s = 210
>>> t = 3.5
>>> s / t
60.0
```

Средняя скорость оказалась равна 60 км/час.

Кроме этого, в интерпретаторе есть режим интерактивной помощи. Для входа в него введите `help()` в интерпретаторе. В режиме помощи можно прочитать справку о всех элементах языка Python: операторах, выражениях, функциях, декораторах, классах и т.д. Для выхода из режима помощи введите `quit`.

Закрыть интерактивный режим можно нажатием клавиш Ctrl + D (Linux) или Ctrl + Z, Enter (Windows).

Установка среды разработки (IDE)

Кроме интерактивного режима, интерпретатор Python поддерживает режим выполнения программ. В этом режиме он выполняет команды, полученные из файла с исходным кодом программы. Для запуска программы из файла нужно выполнить в командной строке

`python` <*имя файла с программой*>

Файлы с программами на Python обычно имеют расширение `.py`. При запуске интерпретатора с ключом `-i` интерпретатор остается в интерактивном режиме после выполнения программы с сохранением всех объектов программы в памяти и возможностью получить к ним доступ; этот режим полезен при отладке небольших скриптов.

Для создания файла с программой достаточно любого текстового редактора (Notepad, Wordpad для Windows; vim, emacs, mcedit для Linux), од-

нако обычно разработка программ (на любом языке) ведется в специализированных *интегрированных средах разработки* (Integrated Development Environments; IDEs). Это программные системы, включающие в себя текстовый редактор с подсветкой синтаксиса кода, средства для запуска компилятора (интерпретатора), отладчик кода, встроенную документацию и др. С интерпретатором Python распространяется простейшая IDE «IDLE». Для ее запуска нажмите Win+R (Windows) или откройте окно терминала (Linux) и выполните команду

```
idle
```

После выполнения команды произойдет запуск интерпретатора Python в интерактивном режиме, дополненном подсветкой синтаксиса и автодополнением кода по клавише ⭾. Для создания нового файла нажмите Ctrl + N (или выберите File › New file в меню); при этом откроется окно текстового редактора. Напишем простейшую программу: в окно редактора вставим

```
1 print "Hello world!"
```

и запустим программу на выполнение, нажав F5. Согласившись сохранить программу в файл, мы получим следующий результат ее выполнения:

```
>>> ========================== RESTART ==========================
>>>
Hello world!
```

`print` в Python 2 — оператор вывода на экран. Однако при попытке изменить строку `Hello world!` на строку, содержащую русские буквы (например, `Привет, мир!`) программа завершится с ошибкой. Это происходит оттого, что по умолчанию интерпретатор Python считает, что программа написана в кодировке ASCII, в которую не входят кириллические символы. Кодировка описывается в первой строке программы:

```
# -*- coding: cp1251 -*-
```

объявляет кодировкой программы CP–1251 (кириллическую Windows). Установка кодировки **необходима**, если в любом месте (даже в комментариях!) файла с программой на Python встречаются русские буквы.

Помимо IDLE, существуют более совершенные и удобные среды разработки на Python. В нашем курсе мы будем использовать PyCharm от JetBrains. Она имеет свободно распространяющуюся версию Community Edition с открытым исходным кодом. Для ее установки скачайте последнюю версию программы с сайта

`https://www.jetbrains.com/pycharm/download/`,

и запустите установщик (в Windows) либо распакуйте архив с программой (в Linux).

Задания

1.2.1. Установите интерпретатор Python на свой компьютер, используя приведенный алгоритм.

1.2.2. Проверьте правильность установки интерпретатора.

1.2.3. Решите в интерпретаторе следующий пример:

$$\sqrt[5]{\left(\frac{1}{2}\right)^7}$$

1.2.4. Определите результаты выполнения в интерпретаторе следующих выражений:

`3/2`, `3./2`, `3./2.`, `3//2`, `3.//2.`

Какие из них одинаковы, какие различны? Почему?

1.2.5. Автомобиль едет по закруглению радиусом 100 метров со скоростью 72 км/час. Чему равно центростремительное ускорение, действующее на автомобиль?

1.2.6. Прочитайте интерактивную справку по операторам **`print`**, **`def`**, `+`.

1.2.7. Напишите программу, решающую задачу из примера 1.1.

1.2.8. Напишите программу, выводящую на экран слова «`Привет, мир!`».

1.2.9. Установите на свой компьютер среду разработки PyCharm.

1.3. Основные элементы языка

Типы переменных

Как мы уже видели, для хранения данных в памяти используются *переменные*. Переменная — это объект Python. У каждого объекта есть три характеристики:

- адрес ячейки в памяти, где хранится объект (может быть получен функцией `id()`);
- тип объекта, определяющий область допустимых значений объекта и набор операций над ним;
- значение — данные, хранящиеся в объекте.

Пример 1.2. Присвоим переменной a значение 2 и рассмотрим ее характеристики:

```
>>> a = 2
>>> id(a)
37053536L     # адрес ячейки
>>> type(a)
<type 'int'> # тип - целое число
>>> a
2             # значение равно 2
```

Адрес и тип объекта определяются при его создании в программе и доступны только для чтения. Возможность изменения значения объекта определяется его типом — в Python существуют как *изменяемые*, так и *неизменяемые* объекты. Подробнее об объектах в Python рассказано в главе 1.4.

Python — язык с сильной динамической неявной типизацией. Рассмотрим подробнее, что это означает.

- В языках с *сильной типизацией* интерпретатор (или компилятор) не производит неявного преобразования типов переменных — например, строковая переменная `"3"` не может стать целым числом `3`, если не приведена к типу `int` явным образом. То есть, выражение `3 + "3"` вызовет ошибку, т.к. операция сложения числа и строки не определена.
- В языках с *динамической типизацией* типы переменных определяются их значениями на этапе выполнения программы; типами обладают не переменные, а значения.
- В языках с *неявной типизацией* типы переменных определяет компилятор — не нужно явно описывать переменные в преамбуле программы.

Кратко рассмотрим т.н. «примитивные» типы переменных в Python.

Числа

В Python существует несколько типов числовых значений:

- Целое число (`int`) — например, `255`.
- Длинное целое число (`long`) — число, большее $2^{32} - 1$. Тип устарел и объединен с `int` в Python 3.
- Дробное число (`float`). Может быть записано как в виде `0.001`, так и в виде `1e-3`. e — экспонента, `12.3e-4` обозначает $12.3 \cdot 10^{-4}$.
- Комплексное число (`complex`). Мнимая часть обозначается `j` — например, `1 + 2j`.

Списки и кортежи

Списки (`list`) и кортежи (`tuple`) — *последовательности* значений. Элементами списков и кортежей могут быть любые объекты. Отличаются они друг от друга тем, что кортеж — неизменяемый тип, а список — изменяемый. Для доступа к элементам последовательностей используются квадратные скобки. Доступ к элементам списка и кортежа производится по индексу начиная с **нуля** (приложение A.4).

Пример 1.3. Создадим список из 4 элементов, преобразуем его в кортеж и попытаемся изменить один из элементов:

```
>>> a = [1, 2, 'string', True]
>>> a[2] = 3
>>> a
[1, 2, 3, True]
>>> b = tuple(a)
>>> b
(1, 2, 3, True)
>>> b[2] = 0
Traceback (most recent call last):
TypeError: 'tuple' object does not support item assignment
```

При попытке изменить элемент кортежа возникает ошибка, говорящая, что объект «кортеж» не поддерживает операцию присвоения элемента.

Для списков и кортежей, как и для других последовательностей, доступны встроенные функции, основные из которых приведены в прил. A.5.

Строки

Строковый тип (`str`) — *последовательность* символов. Строки могут быть как в одинарных ('), так и в двойных (") кавычках — разницы между ними нет:

```
>>> a = 'string'
>>> b = "string"
>>> a == b
True
```

Одинарные кавычки могут использоваться внутри двойных, двойные — внутри одинарных. Кроме этого, Python поддерживает многострочные переменные. Для их ввода используются *тройные* кавычки (одинарные или двойные).

```
>>> a = "Fred's house"
>>> b = 'He said "Hello"'
```

```
>>> c = """This is a string
which expands
over several lines"""
```

Возможные способы доступа к символам строки описаны в приложении А.4. Нумерация символов строки тоже начинается с нуля.

Строка — неизменяемый тип. Это означает, что символ в строке нельзя изменить прямым к нему обращением.

Форматирование строк в Python и вставка в них значений переменных может быть проведена двумя способами: с помощью *форматирующего выражения* (основанного на синтаксисе функции `printf` языка С) и путем вызова метода `format`. Форматирующее выражение строится с помощью оператора %, определенного для строк. Слева от оператора должна стоять строка, в которой на месте подстановки переменных должны быть заполнители, начинающиеся с % (например, `%d`). Формат заполнителя зависит от типа переменной; возможные форматы описаны в приложении А.6. Справа от оператора должен быть кортеж переменных, значения которых подставляются в строку.

Пример 1.4. Выведем красиво результат задачи, разобранной в примере 1.1

```
>>> print 'Mean velocity: %f km/h, distance: %i km' % (s/t, s)
Mean velocity: 60.0 km/h, distance: 210 km
```

Кроме этого, строки в Python могут содержать также *управляющие символы*, начинающиеся с обратного слеша: `'\n'` — перевод строки; `'\t'` — знак табуляции; `'\\'` — обратный слеш; Кроме этого, в Python есть еще несколько типов: логический (`bool`), к которому относятся 2 значения: `True` и `False`; словари (`dict`) — структура для хранения пар «ключ-значение» (`{'name':'Monty', 'age':43}`); множества (`set`) — неизменяемая структура для хранения неупорядоченного множества уникальных объектов (`{1,2,'a',3.14}`); специальный тип `NoneType`, содержащий пустой элемент `None`. Подробнее об этих типах можно прочитать в руководстве по Python.

Операторы и функции

Как в других языках программирования, логика программы описывается последовательностью ее строк. Код программы разбивается на *блоки*, которые определяются `отступами`, т.е. количеством пробелов в начале строки. Строки из одного блока кода должны иметь одно и то же количество пробелов в начале, что делает код более читаемым.

Для контроля потока выполнения программы в Python есть условный оператор **`if ... elif ... else`**, а также операторы циклов **`for`** и **`while`**.

Условный оператор проверяет некоторое условие и в зависимости от результата проверки выполняет один из блоков кода. Синтаксис условного оператора показан в примере.

Пример 1.5. Данный пример вычисляет знак числа a. Степень вложенности блока кода определяется четырьмя пробелами. Ключевое слово **elif** означает `else if` — «иначе если» — и позволяет записать оператор выбора без увеличения вложенности блоков.

```
if a > 0:
    sign = 1
elif a == 0:
    sign = 0
else:
    sign = -1
```

Цикл **for** выполняется для всех элементов некоторой последовательности. Синтаксис цикла приведен ниже:

```
seq = [1, 2, 'string', True]
for i in seq:
    print i
```

Данный пример выводит на экран элементы списка `seq`. Здесь i по очереди принимает значения из набора данных. Часто необходимо провести цикл по числам до некоторого n. Для этого в Python существует функция `range(n)`, создающая список от 0 до $n - 1$. Покажем ее использование на следующем примере, делающем то же, что и предыдущий:

```
seq = [1, 2, 'string', True]
for i in range(len(seq)):
    print seq[i]
```

Любой объект, который может использоваться в цикле **for** `...` **in** `...`, называется *итератором*. Итераторами являются строки (итерации цикла проходят по символам строки), списки и кортежи (по элементам последовательностей), файловые объекты (по строкам файла).

Цикл **while** выполняется, пока верно некоторое условие. Синтаксис:

```
i = 4
while i < 9:
    print i
    i = i + 2
```

Этот цикл выведет на экран числа 4, 6 и 8. Цикл **while** часто используется для того, чтобы создать *бесконечный* цикл — цикл с всегда истинным условием. Для выхода из такого цикла используется оператор **break**. В следующем примере на экран будут выводиться возрастающие числа до тех пор, пока пользователь не введет `q` (`raw_input` — функция, считыва-

ющая строку со стандартного ввода и сохраняющая ее в переменную):

```
i = 0
while True:
    i = i + 1
    print i
    exit = raw_input('Press q to exit, any other key to continue')
    if exit == 'q':
        break
```

В Python есть также оператор **continue**, заканчивающий текущую итерацию цикла.

Кроме циклов, в Python существуют т.н. *списковые включения* — конструкции, создающие новые списки на основе существующих. Синтаксис показан на примере ниже:

```
squares = [x ** 2 for x in range(10)]
```

В этом примере `squares` — список квадратов чисел от 0 до 9.

В *функции* выносится код, который планируется использовать несколько раз. Функции вводятся ключевым словом **def**. Функция может иметь обязательные и необязательные *аргументы*. Необязательные аргументы должны иметь значение по умолчанию. Результат выполнения функции возвращается ключевым словом **return**.

Пример 1.6. В этом примере определена функция `distance`, рассчитывающая расстояние между двумя точками, координаты которых задаются в виде двух списков:

```
1 def distance(crd1, crd2=[0.,0.,0.]):
2     return ((crd1[0] - crd2[0])**2 + (crd1[1] - crd2[1])**2 +
3                (crd1[2] - crd2[2])**2) ** 0.5
```

Вызов функции осуществляется так:

```
>>> distance([1,1,1], [1,1,2])
1.0
```

Если второй список не задан, рассчитывается расстояние от точки до начала координат:

```
>>> distance([2,0,0])
2.0
```

Работа с файлами

Работа с файлами традиционна для языков программирования. Файл — объект, который возвращается встроенной функцией `open`. Возможность чтения или записи в файл регулируется режимом работы с ним, который определяется при открытии файла. Режимы работы с файлами включают

`r` — чтение, `w` — запись в файл, `a` — добавление в файл. Кроме этого, можно явно определить, что файл имеет двоичный формат. Для этого к режиму работы добавляется `b`. Разберем следующие примеры.

Пример 1.7. Мы открываем `file.txt` для чтения и записываем все содержимое файла в переменную `contents`; затем закрываем файл:

```python
f = open('file.txt', 'r')
contents = f.read()
f.close()
```

Пример 1.8. Мы открываем `file.txt` для записи и записываем туда содержимое переменной `string`:

```python
f=open('file.txt', 'w')
f.write(string)
f.close()
```

Кроме `f.read`, есть также метод `f.readlines`, возвращающий список строк файла.

Хорошим тоном считается закрывать все файлы перед завершением программы. Для автоматического закрытия файлов в Python есть оператор `with .. as ..`:

```python
with open('file.txt', 'w') as f:
    f.write(string)
```

Данный код делает то же, что и пример 1.8, но закрывает файл автоматически при выходе из блока `with`.

Следующие два примера показывают запись данных в файл и их чтение.

Пример 1.9. Допустим, нам необходим файл `data.txt` с данными, сформированный следующим образом:

```
File Header
Info    x     y
text of start line
Data   0   0
Data   1   1
Data   2   8
...
text of end line
```

Напишем программу, создающую такой файл.

```python
with open('data.txt', 'w') as f:
    f.write('Header of the file \n')
    f.write('Info       x       y \n')
    f.write('text of start line\n')

    data_string =  'Data %f %f \n'
    for x in range(50):
```

```
        f.write(data_string % (x, x**3))
    f.write('text of end line\n')
```

Пример 1.10. Прочитаем файл, созданный в примере выше. Нам необходимо получить данные, находящиеся между `start line` и `end line`, принимая во внимание,что первый элемент в необходимых строках — строка, нам не нужная. Программа, считывающая данные, приведена ниже.

```
data = []
read_flag = False
with open('data.txt','r') as f:
    for line in f:
        if 'start line' in line:
            read_flag = True
            continue
        if 'end line' in line:
            read_flag = False
            continue
        if read_flag:
            data.append([float(i) for i in line.split()[1:]])
print data
```

Разберем ее построчно. В первых строках инициализируются переменные `data` — список, содержащий данные, и `read_flag` — логческий флаг, показывающий, что при чтении файла мы читаем строки с данными, которые надо разбирать. Далее происходит открытие файла для чтения и цикл, в котором происходит чтение файла построчно. Строки 5–7 — если в строке есть подстрока `start line`, следующие строки будут с данными, и мы устанавливаем флаг. Строки 8–10 — если в строке есть подстрока `end line`, данные закончились, и мы выключаем флаг. Строки 11–12 — разбор данных с помощью спискового включения.

Задания

1.3.1. Напишите функцию, принимающую список и возвращающую кортеж чисел из этого списка, больших 5.

1.3.2. В Python 2 кодировка строк по умолчанию — ASCII. Для того, чтобы в строке можно было использовать русские буквы, необходимо определить строку в кодировке Unicode. Для этого добавим `u` перед началом строки:

```
# -*- coding: utf-8 -*-
print "Привет, мир!"
print u"Привет, мир!"
```

Выполните программу и определите ее результат. Почему он такой?

1.3.3. Напишите функцию, принимающую число n и возвращающую сумму чисел, не больших n, делящихся на 3 и на 5.

1.3.4. Решите предыдущую задачу с помощью спискового включения.

1.3.5. Запишите результат выполнения задачи 1.3.1 в файл. Каждый элемент списка должен быть на новой строке.

1.4. Объекты и классы

До настоящего момента мы рассматривали программы, написанные в императивном (процедурном) стиле, состоящие из наборов инструкций, которые манипулируют данными «сверху вниз», от начала до конца программы. Кроме императивного стиля, существуют другие парадигмы программирования, одну из которых мы рассмотрим в настоящей главе. Она основана на инкапсуляции данных и процедур работы с ними в нечто, называемое *объектом*. Такой стиль программирования называется *объектно-ориентированным* (ООП).

ООП основано на понятиях *класса* и *экземпляра* класса. Класс — **тип** объекта, а экземпляр класса — **объект** этого типа. Объект может хранить данные в переменных, связанных с ним. Такие переменные называются *атрибутами* объекта. Также с объектом могут быть связаны функции, описывающие его поведение. Эти функции называются *методами* объекта. Атрибуты и методы могут быть связаны как с конкретным экземпляром класса, так и с классом в целом. В последнем случае они называются принадлежащими классу.

Класс описывается с помощью ключевого слова `class`. Рассмотрим пример:

```
class Simple(object):
    pass

instance = Simple()
```

Здесь описан класс Simple, наследующий от основного в Python класса object, и создан экземпляр этого класса. Заметим, что ни атрибутов, ни методов в нем не определено (`pass` — пустой оператор).

При создании экземпляра класса вызывается метод `__init__` класса, который может принимать аргументы. В методах класса первым агрументом **обязательно** идет `self` — аргумент, ссылающийся на экземпляр класса. Доступ к атрибутам и методам объекта осуществляется через точку: `self.x` — атрибут `x` данного экземпляра класса.

Пример 1.11. Создадим класс Point, описывающий точку на плоскости:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

```

```
    def distance(self, other):
        return ((self.x-other.x)**2+(self.y-other.y)**2)**0.5

point = Point(2, 3)
```

Класс имеет конструктор `__init__` и метод `distance`. Аргументы метода `distance` — `self`, ссылающийся на данный экземпляр класса, и `other` — другой экземпляр этого же класса. Последней строкой мы создаем экземпляр класса Point с координатами $(2, 3)$ и связываем его с переменной `point`.

Заметим, что в примере мы обращаемся к атрибутам `x` и `y` объектов `self` и `other`. Мы предполагаем, что объект `other` — экземпляр класса `Point`; интерпретатор же не знает ничего об объекте `other`, кроме того, что у него должны быть атрибуты `x` и `y`. Если у `other` не окажется указанных атрибутов, при выполнении программы возникнет ошибка `AttributeError`. Однако, если мы случайно передадим в метод `distance` объект другого типа, имеющий атрибуты `x` и `y`, ошибки не возникнет, однако поведение программы в дальнейшем будет непредсказуемо. Это называется *утиная типизация* — «то, что выглядит, как утка, и крякает, как утка, скорее всего, является уткой» — «любой объект, имеющий атрибуты `x` и `y` — экземпляр класса `Point`». Передача в функции объектов нужного типа возлагается на плечи программиста.

Задания

1.4.1. Создайте два экземпляра класса `Point` и рассчитайте расстояние между ними. Проверьте правильность работы метода `distance`.

1.4.2. Напишите метод `offset` класса `Point`, сдвигающий точку на заданный вектор.

1.4.3. Напишите класс `Vector`, описывающий ветор в трехмерном пространстве. Напишите следующие методы: `add`, складывающий два вектора; `dot` — скалярное произведение двух векторов; `cross` — векторное произведение.

1.5. Элементы функционального программирования

В отличие от императивного и объектно-ориентированного программирования, при использовании которых мы можем изменять объекты программы, функциональный стиль предполагает, что результат работы программы — некая функция $f(x_1 \dots x_n)$, где $x_1 \dots x_n$ — входные параметры. Это означает, что:

1. функции в функциональных языках программирования не имеют *побочных эффектов* — они не изменяют состояния объектов программы;

2. соответственно, не используется оператор присваивания — все необходимые параметры передаются в программу как аргументы функции;
3. в функциональных языках не используются циклы — вместо них используется рекурсия;
4. функции — *объекты первого порядка*, т.е. они ведут себя как обычные объекты — могут быть переданы в другие функции в качестве аргументов и быть возвращены как результат работы других функций.

Хоть Python — не чисто функциональный язык программирования, в нем есть много черт, присущих таким языкам. Рассмотрим некоторые из них.

Анонимные функции

Вследствие того, что функции в Python — объекты первого порядка, существует возможность передавать функции как аргументы в другие функции. В качестве такие аргументов можно использовать `lambda`-выражения — короткие функции, не имеющие имени (в отличие от функций, определяемых ключевым словом `def`). Синтаксис `lambda`-выражения:

```
lambda x, y : x + y
```

где `x`, `y` — список аргументов анонимной функции, а `x + y` — возвращаемое функцией выражение.

Пример 1.12. В Python существует встроенная функция `sorted`, сортирующая список. Эта функция принимает необязательный параметр `key`, значение которого — функция одного аргумента, возвращающая ключ для сортировки. Основное предназначение — сортировать сложные структуры данных по некоторому параметру.

```
>>> students = [
        ('john', 'A', 15),
        ('jane', 'B', 12),
        ('dave', 'B', 10),
]
>>> sorted(students, key=lambda student: student[2])
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

В коде выше `students` — список записей учеников, содержащий имя ученика, его оценку и возраст. Мы сортируем список, взяв в качестве ключа возраст ученика (элемент с индексом 2 в каждой записи).

Пространства имен

Каждый раз, когда вызывается функция, создается новое *пространство имен* — словарь, содержащий названия и значения параметров функции и локальных для функции переменных. Значения всех переменных,

существующих в функции, интерпретатор сначала ищет в пространстве имен, локальных для этой функции; затем — в глобальном пространстве имен (модуле, в котором функция определена); затем — в пространстве встроенных имен. При присвоении переменной значения внутри функции переменная попадает в локальное пространство имен, которое при выходе из функции удаляется. Для того, чтобы изменить глобальную переменную внутри функции, ее надо объявить с ключевым словом `global`.

Пример 1.13. В следующем коде значение переменной a при выходе из функции остается равным 3, хоть она и была изменена внутри функции:

```
a = 3
def f():    # определение функции f
    a = 2
f()         # вызов функции f
print a
```

Пример 1.14. Здесь a становится равным 2.

```
a = 3
def f():    # определение функции f
    global a
    a = 2
f()         # вызов функции f
print a
```

Рекурсия

Рекурсивные функции в Python определяются так же, как в других языках программирования: необходимо определить базовый случай (нерекурсивное завершение функции) и собственно рекурсивный вызов.

Пример 1.15. Данная функция вычисляет факториал числа:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Существует предел на глубину рекурсивных вызовов, по умолчанию равный 1000. Функция `sys.setrecursionlimit()` используется для его изменения, `sys.getrecursionlimit()` — для его просмотра.

Генераторы

Генераторы в Python — объекты, вычисляющие результат до получения некоторого значения, а затем останавливающие вычисления до появления необходимости в следующем значении. В отличие от обычных

функций, генераторы сохраняют свое состояние между вызовами. Определение генератора отличается от определения обычной функции использованием ключевого слова **yield** вместо **return**. При вызове такой функции создается объект-генератор, имеющий методы `next()`, выдающий следующее значение, и `close()`, закрывающий генератор. Генератор в Python является итератором, то есть может использоваться в цикле **for** (см. главу 1.3).

Пример 1.16. Создадим генератор, выдающий факториалы чисел вплоть до n:

```
1 def factorial(n):
2     print 'Calculaing factorials...'
3     result = 1
4     for i in xrange(1, n):
5         result *= i
6         yield result
```

Вызовем его:

```
>>> f = factorial(10)
>>> f
<generator object factorial at 0x0000000001C8EF30>
>>>
```

Видно, что при вызове функции генератора его код не выполняется, а возвращается объект-генератор. Теперь вызовем метод `next()` объекта `f`:

```
>>> f.next()
Calculating factorials...
1
>>> f.next()
2
>>> f.next()
6
```

При вызове `f.next()` начинает выполняться код функции `factorial`, доходит до ключевого слова `yield` и останавливается, запоминая состояние генератора. При следующем вызове `next()` выполнение начинается с того места и с тем окружением, на котором оно прервалось раньше. Используем генератор в цикле:

```
>>> for i in f:
...     print i
24
120
720
5040
40320
362880
```

На 9! генератор заканчивается и при дальнейшем вызове `next()` возникает ошибка `StopIteration`.

Декораторы

Декораторы в Python — по сути «обёртки», дающие возможность каким-то образом изменять результат выполнения декорируемой функции, не изменяя саму функцию. Декоратор определяется как функция, принимающая функцию в качестве аргумента и возвращающая уже «задекорированную» функцию. Применяется он к декорируемой функции с помощью символа `@`:

```
@decorator
def fn():
    ...
```

Пример 1.17. Напишем декоратор, рассчитывающий время выполнения функции. Будем использовать для этого функцию `time.clock()`:

```
import time

def timer(fn):
    def wrapped(*args, **kwargs):
        beg = time.clock()
        res = fn(*args, **kwargs)
        print 'Function ran for {} s'.format(time.clock()-beg)
        return res
    return wrapped
```

Используем этот декоратор для определения времен выполнения функции `distance` из примера 1.6:

```
@timer
def distance(crd1, crd2=[0.,0.,0.]):
    ...
```

Рассчитаем расстояние между (1,1,1) и (2,3,5):

```
>>> distance([1,1,1], [2,3,5])
Function ran for 3.11925921324e-05 s
4.58257569495584
```

Задания

1.5.1. Напишите рекурсивную функцию, вычисляющую n-е число Фибоначчи, рассчитывающееся по формуле

$$F_n = F_{n-1} + F_{n-2}, \quad (1.1)$$

где $F_1 = F_2 = 1$.

1.5.2. Напишите генератор для чисел Фибоначчи.

1.6. Модули

В Python реализована концепция модульного подхода к программированию. Модули позволяют один и тот же код использовать в нескольких программах. *Модуль* языка Python — файл с расширением `.py`, в котором содержатся описания объектов языка (переменных, функций, классов и т.д.). В Python существует большая *стандартная библиотека* модулей, каждый из которых решает отдельную задачу. Кроме этого, пользователями было написано большое количество собственных модулей, более 60000 из которых доступно на сайте Python Package Index (`http://pypi.python.org`).

Для упрощения написания программ Python позволяет создавать собственные модули, которые затем можно объединят в *пакеты* — коллекции модулей. В простейшем случае файл собственного модуля должен находиться в директории с программой; кроме этого, его можно перенести в поддиректорию. При этом в поддиректории должен находиться файл `__init__.py`.

Для использования модуля он должен быть *импортирован* в программу ключевыми словами `import` или `from .. import ...`

Пример 1.18. Здесь мы импортируем модуль `os` (для работы с вызовами операционной системы) и выводим значение переменной `name` (название ОС), определенной в этом модуле.

```
>>> import os
>>> os.name
'nt'
```

Модуль может быть импортирован в программу различными способами, которые сведены в таблицу 1.1.

Пример 1.19. Создадим модуль `point`, содержащий класс Point из примера 1.11. Для этого:

1. код из примера 1.11 сохраним в файл `point.py`;
2. в той же директории, что и `point.py`, создадим файл `program.py` со следующим кодом:

```
1 import point
2 p1 = point.Point(2, 3)
3 p2 = point.Point(2, 4)
4 print p1.distance(p2)
```

При запуске `program.py` импортируется модуль `point.py`, и все функции и классы из него становятся доступны в пространстве имен `point`. Мы создаем два экземпляра класса Point с различными координатами, и считаем декартово расстояние между ними.

Для поиска модулей, решающих конкретные задачи, можно обращать-

Таблица 1.1.: Загрузка модулей различными способами.

Загрузка модуля или функции из модуля	Использование	Описание
`import datetime`	`datetime.date.today()`	Доступ к объектам в модуле — через название модуля
`import datetime as dt`	`dt.date.today()`	Изменяем имя, с которым связан модуль
`from datetime import *`	`date.today()`	Импорт **всех** объектов из модуля по именам (не рекомендуется)
`from datetime import date`	`date.today()`	Импорт **одного** объекта из модуля по имени
`from datetime import date as data`	`data.today()`	Изменяем имя, с которым связан импортируемая функция

ся к документации Python. Установка сторонних модулей может производиться по разному в зависимости от операционной системы.

Windows

Для установки сторонних модулей в Miniconda включен пакетный менеджер `conda`, который связывается с центральным репозиторием, скачивает нужный пакет и устанавливает его. Чтобы его использовать, откройте командную строку (нажмите Win + R и выполните `cmd`).

Разберем следующий пример — установим модуль `numpy` для матричных вычислений. Для того, чтобы убедиться, что модуль есть в репозитории, выполним

```
C:\Users\Andrey>conda search numpy
Fetching package metadata: ....
numpy                    1.6.2             py27_0  defaults
                         1.6.2             py26_0  defaults
...
```

Команда вернет версии модуля, существующие в репозитории для разных версий интерпретатора. Для установки запустите `conda` с ключом `install`; менеджер автоматически выберет нужную версию и предложит установить зависимости (если они есть):

```
C:\Users\Andrey>conda install numpy
Fetching package metadata: ....
```

```
...
The following packages will be downloaded:
    package                    |            build
    ---------------------------|-----------------
    numpy-1.9.2                |           py27_0        23.2 MB
    conda-3.15.1               |           py27_0         214 KB
    ------------------------------------------------------------
                                           Total:        23.4 MB
The following NEW packages will be INSTALLED:
    numpy: 1.9.2-py27_0
The following packages will be UPDATED:
    conda: 3.14.1-py27_0 --> 3.15.1-py27_0

Proceed ([y]/n)?
```

После нажания Enter менеджер автоматически загрузит и установит необходимый пакет со всеми зависимостями. После установки пакет будет доступен для импортирования.

Linux

В Linux установка модулей может быть произведена либо с помощью пакетного менеджера (`yum`, `dnf`, `apt-get`) в дистрибутиве (см. главу 1.2), либо используя команду `pip` в составе Python. Эта команда (запускаемая с правами администратора) служит для установки модулей из `PyPI`. Синтаксис команды прост: `pip search` ищет модуль в `PyPI`, а `pip install` его устанавливает. В качестве примера установим модуль `numpy`.

```
root@host ~ # pip search numpy
...
numpy             - NumPy: array processing for numbers, strings,
                    records, and objects.
...
root@host ~ # pip install numpy
Downloading/unpacking numpy
...
Successfully installed numpy
```

Задания

1.6.1. Установите модули `numpy`, `scipy`, `matplotlib`. Проверьте их установку, импортировав их в интерактивном режиме.

1.6.2. Напишите свой модуль, содержащий функцию, решающую задачу 1.3.3. Используйте его из другой программы.

2. Научные вычисления в Python

2.1. Numpy. Общая информация

Numeric Python (Numpy) — это несколько модулей для поддержки больших многомерных массивов и матриц, а также большая библиотека высокоуровневых математических функций для операций с этими массивами. Numpy необходим для многих численных приложений и активно используется другими модулями.

Массив — это набор однородных элементов, доступных по индексам. Массивы могут быть многомерными, то есть иметь более одной размерности. Количество размерностей и длина массива по каждой оси называются формой массива (shape).

Особо подчеркнём отличие массива от стандартного в Python набора данных (списка или кортежа):

- величины, входящие в массив, имеют одинаковый **тип**;
- количество элементов массива жёстко задаётся при инициализации.

При хранении элементов в списке интерпретатор Python хранит информацию о типе для *каждого элемента* списка; кроме этого, при поэлементных операциях ему необходимо запускать код, ответственный за выбор типа операции для каждого элемента. Массивы позволяют экономить память и увеличивать скорость работы с большим количеством однотипных данных.

Как представлены массивы в Python? В Python массивы — это объекты, содержащие буфер данных и информацию о форме, размерности, типе данных и т.д. Как и у любого объекта, у массива a можно менять атрибуты напрямую: `a.shape = (2, 3)` или через вызов функции `np.reshape(a,(2,3))` (reshape — изменение формы). Такая же ситуация и с методами класса `ndarray`: многие из них имеют соответствующие функции Numpy: `np.resize(a,(2,4))` (resize – изменение размера). Некоторые методы таких функций не имеют, например, `a.flatten`. Для правильного использования таких функций необходимо обращаться к их описанию.

В Numpy реализовано много операций для работы с массивами:

- создание, модификация массива (изменение формы, транспонирование, поэлементные операции);
- выбор элементов;

- операции с массивами (различные типы умножения), сравнение массивов;
- решение задач линейной алгебры (системы линейных уравнений, собственные вектора. собственные значения);
- создание наборов случайных данных;
- и т.д.

Установка Numpy

Установка пакета Numpy происходит аналогично установке Python, описанной в разделе 1.2. Стандартный установщик можно скачать с официального сайта:

`http://www.scipy.org/install.html`.

Можно использовать научные дистрибутивы, описанные в разделе 1.2 «Установка Python», которые уже включают Numpy.

Чтобы использовать Numpy, как и любой модуль Python, его надо сначала импортировать:

```
import numpy as np
```

Тогда использовать его можно с использованием префикса `np`:

```
a = np.arange(6, dtype=np.int)
```

Внимание! Не рекомендуется импортировать все содержимое модуля при помощи команды `from numpy import *`, т.к. это может привести к путанице в именах функций. Лучше использовать общепринятое короткое название модуля `np`.

2.2. Массивы Numpy

В Numpy можно выделить несколько видов массивов:

- произвольные многомерные массивы (array);
- матрицы (matrix) — двухмерные массивы, для которых дополнительно определены операции возведения в степень и перемножения. Для работы с матрицами можно вместо `numpy` подключать `numpy.matrix`, в котором реализованы те же самые операции, только массивы–результаты операций будут приводится к типу `matrix`.
- массивы с маской (masked arrays) — массивы, которые могут содержать пропущенные или неверные значения;

- массивы записей (record arrays) — массивы с возможностью доступа к элементам через атрибуты («столбцы данных»), как в электронной таблице.

Создание массивов из имеющихся данных

Для создания массивов существует множество способов. Самый простой из них — преобразование списка или кортежа при помощи функции `array()`. Доступ к элементам массива производится так же, как и для других последовательностей (приложение A.4):

```
>>> a = np.array([1, 4, 5, 8], float32)
>>> a
array([1., 4., 5., 8.])
>>> type(a)
<type 'numpy.ndarray'>
>>> a[:2]
array([1., 4.])
>>> a[3]
8.0
```

Массивы, как и списки, могут иметь много измерений:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], dtype='float32')
>>> a
array([[1., 2., 3.],
       [4., 5., 6]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
>>> a.shape
(2,3)
```

Для элементов массива в Numpy определено множество типов (взятых из языка C):

- `bool8` — логический;
- `int8(16,32,64)` — целый;
- `uint8(16,32,64)` — целый без знака (натуральный);
- `float16(32,64)` — действительный;
- `complex64(128)` — комплексный.

Число после названия типа указывает количество бит, отводимое под размещение одного элемента данных. Допускается использование типов Python: `bool_`, `int_`, `float_`, `complex_`, `str_`, `object_` — объект Python. В этом случае под элемент отводится столько байт, сколько предусмотрено интерпретатором Python. Размещение массива в памяти прово-

дится в соответствии с опциями и может быть выполнено как в языке C (по последнему индексу) или как в языке Fortran (по первому индексу); для этого используется параметр `order`, принимающий значения `"C"` или `"F"` соответственно.

Присваивание массивов не ведёт к их копированию, создаются ссылки на тот же самый объект. Создание копий должно быть явным:

```
>>> a = np.array([1, 2, 3], dtype='float32')
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

Дополнительные способы создания массивов из имеющихся данных (файлов, строк, буфера в памяти) приведены в приложении B.1

Создание массивов определённого вида

Кроме создания массивов из других объектов с помощью функции `np.array`, существуют различные способы создать массив определённого вида: нулевой (`np.zeros` или `np.zeros_like`), из единиц (`np.ones` или `np.ones_like`), неинициализированный (`np.empty`, `np.empty_like`):

```
>>> a = np.zeros(3)
>>> a
array([ 0., 0., 0.])
>>> b = np.ones_like(a)
>>> b
array([ 1., 1., 1.])
>>> c = np.ones(4)
>>> c
array([ 1., 1., 1., 1.])
>>> d = np.zeros_like(c)
>>> d
array([ 0., 0., 0., 0.])
```

Другие способы создания массивов приведены в приложении B.2.

Создание сеток

Важным для приложений видом массивов являются *сетки*. Они используются для получения в изображений многомерных функций. Многомерные функции хранятся и используются в компьютере в виде массива ее значений, вычисленных на некоторой сетке узлов. Важно использовать именно сетку, т.е. кроме положения каждого узла (координат точки) необходимо указать в ней связи (номера узлов, расположенных рядом). Это необходимо для построения гладких функций, которое возможно после выполнения интерполяции функции на точки, не совпадающие с узлами.

Наиболее простой способ задания сеток, имеющих простую 3D топологию связей (6 соседей для каждого узла), заключается в задании связей через индексы трехмерного массива $\vec{R}(i,j,k)$. В такой топологии узел (i,j,k) находится между узлами $(i-1,j,k)$ и $(i+1,j,k)$ по первому направлению, между узлами $(i,j-1,k)$ и $(i,j+1,k)$ по второму направлению и т.д. Координаты узла задаются значениями в массиве $\vec{R}$. Часто один массив $\vec{R}(i,j,k)$ для удобства заменяют тремя массивами меньшей размерности $X(i,j,k)$, $Y(i,j,k)$, $Z(i,j,k)$. Обратите внимание, что для сетки параллельной осям координат многие значения в массивах X,Y,Z повторяются, но для сетки расположенной произвольно они независимы.

Для задания сеток используется функция `meshgrid()`, принимающая одномерные массивы точек (вдоль каждого измерения) и формирующая многомерную ортогональную сетку:

```
>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = meshgrid(x, y)
>>> xv
array([[ 0. ,  0.5,  1. ],
       [ 0. ,  0.5,  1. ]])
>>> yv
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.]])
>>> xv, yv = meshgrid(x, y, sparse=True)
>>> xv
array([[ 0. ,  0.5,  1. ]])
>>> yv
array([[ 0.],
       [ 1.]])
```

Для получения трехмерных сеток можно использовать конструкцию `np.mgrid`:

```
>>> x, y, z = np.mgrid[0:5:20j, 0:5:0.3, 0:5]
```

Вдоль каждой оси задаются крайние точки разбиения (0 и 5 в данном примере) и параметр разбиения: 20j — разбиение на 20 интервалов; 0.3 — разбиение с шагом 0,3 (последний узел 0 + 0,3n не превышает 5); без параметра — шаг равен 1. Обратите внимание, что `np.mgrid` — это не функция, а объект, у которого берутся срезы.

Используя функции для векторов из модуля Numpy, легко задать аналитическую функцию на этой сетке:

```
x, y, z = np.mgrid[0:5:20j, 0:5:0.3, 0:5].
V3d = np.cos(10*x) + np.cos(10*y) + np.cos(10*z) +
        2 * (x**2 + y**2 + z**2)
```

Массив $V3d$ имеет размеры, равные размеру сетки, и хранит значения, соответствующие каждому узлу. Это позволяет создавать легко читаемые программы и оптимальным образом проводить вычисления.

Другие способы создания сеток приведены в приложении B.3.

Трансформации массива без изменения элементов

В Numpy создано множество методов для преобразования существующих массивов: изменение формы, разбиение массива на части, транспонирование и т.д. Самые распространённые функции собраны в приложениях B.4 и B.5. Например, можно изменить форму массива:

```
>>> a = np.array(range(10), float32)
>>> a
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> b = a.reshape((5,2))
array([[ 0., 1.],
[ 2., 3.],
[ 4., 5.],
[ 6., 7.],
[ 8., 9.]])
>>> b.shape
(5, 2)
```

Операции с массивами

Поэлементные операции с массивами делаются тривиально:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
```

```
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

При несовпадении форм массивов происходит автоматическая попытка приведения более простой формы к более сложной:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1., 2.],
[ 3., 4.],
[ 5., 6.]])
>>> b
array([-1., 3.])
>>> a + b
array([[ 0., 5.],
[ 2., 7.],
[ 4., 9.]])
```

Можно контролировать приведение форм с помощью `np.newaxis`:

```
>>> a = np.zeros((2,2), float)
array([[ 0., 0.],
[ 0., 0.]])
>>> b = np.array([-1., 3.], float)
array([-1., 3.])
>>> a + b[np.newaxis,:]
array([[-1., 3.],
[-1., 3.]])
>>> a + b[:,np.newaxis]
array([[-1., -1.],
[ 3., 3.]])
```

Функции, определенные для массивов

В Numpy есть большая библиотека математических функций, которые могут применяться поэлементно к массивам: abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh.

```
>>> a = np.linspace(0.3,0.6,4)
array([ 0.3, 0.4, 0.5, 0.6, 0.7])
```

```
>>> np.sin(a)
array([ 0.29552021, 0.38941834, 0.47942554,0.56464247])
```

Полное собрание функций можно найти на сайте:

http://docs.scipy.org/doc/numpy/reference/routines.math.html.

Самые используемые алгебраические и тригонометрические функций даны в приложениях B.6 и B.7. Бинарные функции (функции двух аргументов) перечислены в приложении B.8. Для правильной работы с логическими бинарными функциям (`and`, `or`) необходимо явно их записывать через функции модуля Numpy.

Дополнительно в приложении B.9 приведены функции, позволяющие проводить вычисления для массивов при помощи функций, написанных пользователем.

Сортировка, поиск, статистика

Функции, позволяющие сортировать массив, осуществлять поиск максимального/минимального элемента, определять его положение в массиве, приведены в приложении B.10. В приложении B.11 приведены основные функции для получения статистики по данным в массиве. Сюда относятся средние значения, построение распределений (гистограмм), построение матрицы ковариаций и т.д.

Пример 2.1. Рассчитаем среднее, дисперсию, стандартное отклонение для массива a:

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

Можно выбрать ось массива для маржинальной статистики:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2., 2.])
>>> a.mean(axis=1)
```

```
array([ 1., 1., 4.])
>>> a.min(axis=1)
array([ 0., -1., 3.])
>>> a.max(axis=0)
array([ 3., 5.])
```

Задания

2.2.1. Получите доступ к пакету Numpy под именем `np`.
Создайте нулевой вектор длины 10.
Создайте вектор длины 10, у которого каждый третий элемент, начиная со второго, равен 1.

2.2.2. Создайте вектор с элементами от 10 до 49.
Создайте вектор длины 10 с равноотстоящими элементами от 0 до 1, не включая 0 и 1.

2.2.3. Создайте единичную матрицу 3x3.
Создайте матрицу 3x3 с элементами от 0 до 8.

2.2.4. Создайте матрицу 5x5 с элементами [1,2,3,4] под главной диагональю.
Создайте матрицу 5x5 с одинаковыми рядами [1,2,3,4,5]

2.2.5. Создайте матрицу 8x8 и заполните ее как шахматную доску.
Создайте шахматную доску 8x8 с помощью функции np.tile.

2.2.6. Создайте массив с элементами [1,2,0,0,4,0] и найдите индексы его ненулевых элементов.

2.2.7. Создайте массив 10x10 со значениями $\cos(i + j^2)$ и найдите ее максимум и минимум.
Обнулите 3 наибольших и 3 наименьших элемента массива.

2.2.8. Создайте одномерный массив длины 30 со значениями $\sin(i)$.
Отнормируйте его.
Отсортируйте его.
Найдите среднее значение его элементов.

2.3. Дополнительные модули NumPy

Дискретное преобразование Фурье (numpy.fft)

Для преобразования Фурье доступен ряд алгоритмов, реализованных в модуле `numpy.fft` и перечисленных в приложении B.12. Это прямые и обратные преобразования Фурье для действительных и комплексных

функций. Дополнительные функции для работы со спектрами можно найти в модуле `scipy.fftpack` пакета Scipy, описанном в разделе 2.4.

Пример 2.2. Преобразование Фурье от одномерной функции $\sin(t)$. Задаем значение функции в точках, находим коэффициенты Фурье и частоты, которым они соответствуют. Строим графики (см. гл. 2.6).

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> t = np.arange(256)
>>> f = np.sin(t)
>>> sp = np.fft.fft(f)
>>> freq = np.fft.fftfreq(t.shape[-1])

>>> import matplotlib.pyplot as plt
>>> plt.plot(freq, sp.real, freq, sp.imag)
>>> plt.show()
```

Линейная алгебра (numpy.linalg)

Модуль `numpy.linalg` содержит алгоритмы линейной алгебры:

- нахождение определителя матрицы;
- решение системы линейных уравнений;
- обращение матрицы;
- нахождение собственных чисел и собственных векторов матрицы;
- разложение матрицы на множители: Холецкого, сингулярное;
- метод наименьших квадратов;
- и т.д.

Краткий обзор функций дан в приложении B.13. Дополнительные функции для работы с обычными и разреженными матрицами можно найти в модуле `scipy.linalg` пакета Scipy, описанном в разделе 2.4.

Пример 2.3. Решим систему линейных уравнений:

$$\begin{cases} 3 \cdot x_1 + 1 \cdot x_2 = 9 \\ 1 \cdot x_1 + 2 \cdot x_2 = 8 \end{cases}$$

```
>>> import numpy as np
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])

>>> np.allclose(np.dot(a, x), b)
True
```

Пример 2.4. Простой пример нахождения собственных значений и векторов:

```
>>> import numpy as np
>>> w, v = np.linalg.eig(np.diag((1, 2, 3)))
>>> print w, v
array([ 1.,  2.,  3.])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Значения могут получится комплексными:

```
>>> import numpy as np
>>> w, v = np.linalg.eig(np.array([[1, -1], [1, 1]]))
>>> print w, v
array([ 1. + 1.j,  1. - 1.j])
array([[ 0.70710678+0.j          ,  0.70710678+0.j          ],
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]])
```

Не забывайте про численные ошибки. В следующем примере собственные значения должны быть равны $1 + 10^{-9}$ и $1 - 10^{-9}$, а при вычислениях получается 1.

```
>>> import numpy as np
>>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
>>> w, v = np.linalg.eig(a)
>>> print w, v
array([ 1.,  1.])
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Случайные величины (numpy.random)

В модуле `numpy.random` собраны функции для генерации массивов случайных чисел различных распределений и свойств. Их можно применять для математического моделирования. Функция `random()` создает массивы из псевдослучайных чисел, равномерно распределенных в интервале (0, 1). Функция `randint()` для получения массива равномерно распределенных чисел из заданного интервала и заданной формы. Можно получать и случайные перестановки с помощью `permutation()`. Доступны и другие распределения для получения массива нормально распределенных величин с заданным средним и стандартным отклонением и т.д. Основные функции приведены в приложении B.14. Дополнительные возможности для работы с различными случайными распределениями и работы со статистической информацией можно найти в модуле `scipy.stats` пакета Scipy, описанном в разделе 2.4.

Пример 2.5. Получение равномерно распределенных случайных чисел:

```
>>> import numpy as np
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],
       [ 0.37601032,  0.25528411],
       [ 0.49313049,  0.94909878]])
```

Пример 2.6. Набор случайных данных со стандартным нормальным распределением (параметры (0,1)). Если нужно получить нормальное распределение с параметрами (μ, σ^2), можно использовать формулу $\sigma \cdot \text{np.random.randn}(\ldots) + \mu$.

```
>>> import numpy as np
>>> np.random.randn()
2.1923875335537315

>>> 2.5 * np.random.randn(2, 4) + 3  # mu = 3, sigma = 6.25
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])
```

Полиномы (numpy.polynomial)

Модуль полиномов обеспечивает стандартные функции работы с полиномами разного вида. В нем реализованы полиномы Чебышева, Лежандра, Эрмита, Лагерра. Для полиномов определены стандартные арифметические функции +, -, *, //, деление по модулю, деление с остатком, возведение в степень и вычисление значения полинома. Важно задавать область определения, т.к. часто свойства полинома (например, при интерполяции) сохраняются только на определенном интервале.

В зависимости от класса полинома сохраняются коэффициенты разложения по полиномам определенного типа, что позволяет получать разложение функций в ряд по полиномам разного типа. Функции для работы с полиномами (разложение по полиномам, вычисление производных/интегралов от полиномов и т.д.) приведены в приложении B.15. Некоторые функции (например, интерполяция данных) возвращают объект типа полином. У этого объекта есть набор методов, позволяющих извлекать и преобразовывать данные. Методы для полиномов перечислены в том же приложении.

Пример 2.7. Ниже дан пример использования полиномов. Обратите внимание, что у полиномов есть окно (window) и домен (domain). Домен задает область аргументов функции при вызове, а окно показывает область аргументов «классического» полинома, которая смещается и масштабируется на домен. Полиномы с разным окном или доменом, а также разного типа не могут одновременно участвовать в арифметических операциях. При операциях, содержащих полиномы и списки, кортежи или массивы, последние преобразуются в полиномы с нужным окном, доменом и типом.

```
>>> from numpy.polynomial import Polynomial as P
```

```
>>> p = P([1, 2, 3])

>>> p + p
Polynomial([ 2., 4., 6.], [-1., 1.], [-1., 1.])

>>> p * p
Polynomial([ 1., 4., 10., 12., 9.], [-1., 1.], [-1., 1.])

>>> quo, rem = divmod(p, P([-1, 1]))
>>> quo
Polynomial([ 5., 3.], [-1., 1.], [-1., 1.])
>>> rem
Polynomial([ 6.], [-1.,  1.], [-1.,  1.])

>>> p(np.arange(5))
array([ 1., 6., 17., 34., 57.])

>>> p(p)
Polynomial([ 6., 16., 36., 36., 27.], [-1., 1.], [-1., 1.])

>>> p.roots()
array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
```

Пример 2.8. Преобразование полиномов разных типов друг в друга осуществляется через метод `convert`:

```
>>> from numpy.polynomial import Polynomial as P
>>> from numpy.polynomial import Chebyshev
>>> p = P.fromroots([1, 2, 3])
>>> p
Polynomial([ -6., 11., -6., 1.], [-1., 1.], [-1., 1.])
>>> p.convert(kind=Chebyshev)
Chebyshev([ -9., 11.75, -3., 0.25], [-1., 1.], [-1., 1.])
```

Пример 2.9. Полиномиальная интерполяция данных осуществляется довольно просто. Ниже показана интерполяция зашумленного синуса полиномами Чебышева. Строим графики исходной функции и полинома.

```
>>> import numpy as np
>>> from numpy.polynomial import Chebyshev as T
>>> np.random.seed(11)
>>> x = np.linspace(0, 2*np.pi, 20)
>>> y = np.sin(x) + np.random.normal(scale=.1, size=x.shape)
>>> p = T.fit(x, y, 5)
>>> p.domain
array([ 0.        ,  6.28318531])
>>> p.window
array([-1.,  1.])

>>> import matplotlib.pyplot as plt
```

```
>>> plt.plot(x, y, 'o')
>>> xx, yy = p.linspace()
>>> plt.plot(xx, yy, lw=2)
>>> plt.show()
```

Задания

2.3.1. Создайте матрицу 10x10 со случайными значениями и найдите ее собственные векторы и собственные значения.
Проверьте результат.

2.3.2. Задайте случайную матрицу A размером 5x5 и случайный вектор b размером 5.
Решите систему линейных уравнений $Ax = b$.
Проверьте результат.

2.3.3. Задайте случайные матрицы 5x3 на 3x2 и перемножьте их.

2.3.4. Задайте на плоскости случайным образом 3 точки.
Найдите площадь треугольника с вершинами в этих точках.

2.3.5. Задайте в пространстве случайным образом 4 точки.
Найдите объем тетраэдра с вершинами в этих точках.

2.3.6. Создайте два одномерных массива длины 100 со значениями аргумента $0.1 \cdot i$ и функции $\sin(0.1i)$.
Подберите полином 5 степени, аппроксимирующий полученную дискретную функцию.
Найдите максимальное отклонение полинома от точек дискретной функции.

2.4. Пакет Scipy

Scientific Python (Scipy) — это пакет, состоящий из множества модулей для выполнения научных вычислений.

Установка

Установка пакета Scipy происходит аналогично установке Python и Numpy, описанной в разделах 1.2, 2.1. Стандартный установщик можно скачать с официального сайта:

`http://www.scipy.org/install.html`.

Можно использовать научные дистрибутивы, описанные в разделе «Установка Python» 1.2, которые уже включают модуль Scipy.

Чтобы использовать любой модуль Python, его надо сначала импортировать:

```
import scipy as sp
```

Внимание! Не рекомендуется импортировать все содержимое модуля при помощи команды `from scipy import *`, т.к. это может привести к путанице в именах функций. Лучше использовать общепринятое короткое название модуля *sp*.

Обработка дискретных данных

В модуле для интерполяции набора данных (scipy.interpolate) реализованы процедуры интерполяции функций одной и многих переменных кусочными функциями, сплайнами, сглаживающими сплайнами и т.д.

Пример 2.10. Интерполяция функций одной переменной. Задаем точки по оси x от 0 до 10 и значение функции $y = \cos(x^2/9)$.

```
>>> import numpy as np
>>> from scipy.interpolate import interp1d

>>> x = np.linspace(0, 5, num=11, endpoint=True)
>>> y = np.cos(-x**2/18.0)
>>> f = interp1d(x, y)
>>> f2 = interp1d(x, y, kind='cubic')
```

Функции `f`, `f2` — линейная и кубическая интерполяции. Функции определены для массивов и их можно вызвать:

```
>>> xnew = np.linspace(0, 5, num=21, endpoint=True)
>>> print f(xnew)
[ 1.          0.99845838  0.99691675  0.99537513  0.99383351
  0.97108755  0.94834159  0.92559563  0.90284967  0.81221283
  0.72157599  0.63093915  0.54030231  0.35385005  0.16739779
 -0.01905446 -0.20550672 -0.38776657 -0.57002643 -0.75228628
 -0.93454613]
>>> print f2(xnew)
[ 1.          0.99719222  0.9968999   0.996616    0.99383351
  0.98597152  0.97015369  0.94342981  0.90284967  0.84542149
  0.76798725  0.66734738  0.54030231  0.38506517  0.20550007
  0.00688383 -0.20550672 -0.42400837 -0.63141229 -0.80812328
 -0.93454613]
```

Кроме этого, можно построить график (см. гл. 2.6).

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
>>> plt.legend(['data', 'linear', 'cubic'], loc='best')
>>> plt.show()
```

Модуль для дискретного преобразования Фурье (scipy.fftpack) — набор оптимизированных функций для дискретного преобразования Фурье для одно, двух, и многомерных функций, также есть косинус-преобразование Фурье. В модуль включены некоторые дифференциальные операторы для фурье-образов функций и функции, помогающие работать с рядами Фурье, например, сдвигающие нулевую гармонику в середину рассматриваемого интервала.

Пример 2.11. Задаем N точек с шагом T по оси Ox и вычисляем функцию y — сумму двух синусов с разными частотами и амплитудами. Находим частоты `Xf` и фурье-образ `Yf`.

```
>>> from scipy.fftpack import fft
>>> N = 600
>>> T = 1.0 / 800.0
>>> x = np.linspace(0.0, N*T, N)
>>> y = np.sin(50.0 * 2.0*np.pi*x) + \
        0.5*np.sin(80.0 * 2.0*np.pi*x)

>>> xf = np.linspace(0.0, 1.0/(2.0*T), N/2)
>>> yf = fft(y)
```

Можно построить график.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(xf, 2.0/N * np.abs(yf[0:N/2]))
>>> plt.grid()
>>> plt.show()
```

В модуль обработки сигналов (scipy.signal) входят различные виды фильтров, сплайнов; вейвлет-анализ и т.д.

Работа с изображениями возможна через модуль scipy.ndimage. В нем реализованы фильтры Гаусса, Лапласа, сплайн-фильтры, фильтры, основанные на преобразовании Фурье и др. Также имеется возможность осуществлять аффинные преобразования, повороты, измерения центра масс, положения максимума и минимума, нахождение средних значений, исследование морфологии и т.д. Возможна загрузка изображения из файла.

Модуль для ввода-вывода (scipy.io) осуществляет запись и чтение файлов форматов MATLAB, Matrix Market, Arff, Netcdf, звуковых файлов Wav, и др.

Математический анализ

Модуль, выполняющий различные виды интегрирования (scipy.integrate). В модуль входит интегрирование функций: вычисление определенных интегралов и двойных, тройных интегралов разными методами. Интегрирование можно проводить для функций, заданных аналитически (через оператор `def()`), так и таблично (в дискретных точках). Кроме этого, в модуле есть функции для интегрирования системы обыкновенных дифференциальных уравнений (решения задачи Коши).

Пример 2.12. Вычислим интеграл $I(a,b) = \int_0^1 (ax^2 + b)dx$. Определяем подынтегральную функцию с параметрами. Вычисляем интеграл I. Второй параметр результата — ошибка интегрирования.

```
>>> from scipy.integrate import quad
>>> def integrand(x, a, b):
...     return a * x*x + b
>>> a = 2
>>> b = 1
>>> I = quad(integrand, 0, 1, args=(a,b))
>>> print I
(1.6666666666666667, 1.8503717077085944e-14)
```

Можно задавать бесконечные пределы интегрирования (параметр inp из модуля NumPy). Вычислим интеграл $I(a,b) = \int_0^\infty a * e^{-b*x} dx$

```
>>> import numpy as np
>>> from scipy.integrate import quad
>>> def integrand(x, a, b):
...     return a * np.exp(-b*x)
>>> a = 2
>>> b = 1
>>> I = quad(integrand, 0, np.inf, args=(a,b))
>>> print I
(1.9999999999999998, 1.1685212802470784e-10)
```

Пример 2.13. Решение задачи Коши для интегрирования обыкновенных дифференциальных уравнений при помощи функции `odeint()` рассмотрено в разделе 3.2.

Модуль для проведения оптимизации scipy.optimize нужен для поиска корней и экстремумов различных функций. Нахождение минимумов возможно через разные алгоритмы для одномерных и многомерных функций, реализован поиск минимумов с ограничениями и глобальных минимумов. В модуле есть подбор функций методом наименьших квадратов, поиск корней одномерных и многомерных функций линейными и нелинейными методами.

Пример 2.14. Поиск минимума фукнции Розенброка, которая используется для оценки

производительности алгоритмов оптимизации:

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2\right].$$

Считается, что поиск глобального минимума для данной функции является нетривиальной задачей.

```
>>> import numpy as np
>>> from scipy.optimize import minimize

>>> def rosen(x):
...     """The Rosenbrock function"""
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

>>> x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
>>> res = minimize(rosen, x0, method='nelder-mead',
...                options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571

>>> print(res.x)
[ 1.  1.  1.  1.  1.]
```

Пример 2.15. Подбор параметров функции заданного вида для минимизации отклонения от набора данных. Задаем функцию func с параметрами и набор данных (xdata, ydata). Значения функции имеют случайный разброс относительно точной функции. После подгонки получаем параметры функции и матрицу ковариации.

```
>>> import numpy as np
>>> from scipy.optimize import curve_fit
>>> def func(x, a, b, c):
...     return a * np.exp(-b * x) + c

>>> xdata = np.linspace(0, 4, 50)
>>> y = func(xdata, 2.5, 1.3, 0.5)
>>> ydata = y + 0.2 * np.random.normal(size=len(xdata))

>>> popt, pcov = curve_fit(func, xdata, ydata)
>>> print popt
[ 2.49980521  1.24723067  0.5027052 ]
>>> print pcov
[[ 0.01775805  0.00673288 -0.0011017 ]
 [ 0.00673288  0.0218761   0.0068855 ]
 [-0.0011017   0.0068855   0.00386998]]
```

Пример 2.16. Решение нелинейной системы уравнений:

$$\begin{cases} F_1 = cos(x_1) + x_4 - 1 \\ F_2 = cos(x_2) + x_3 - 2 \\ F_3 = cos(x_3) + x_2 - 3 \\ F_4 = cos(x_4) + x_1 - 4 \end{cases}$$

Задаем векторную функцию от векторного аргумента. Проводим поиск корней, используя в качестве начального выбора [1,1,1,1].

```
>>> def F(x):
...     return np.cos(x) + x[::-1] - [1, 2, 3, 4]
>>> import scipy.optimize as optimize
>>> x = optimize.broyden1(F, [1,1,1,1], f_tol=1e-14)
>>> x
array([ 4.04674914,  3.91158389,  2.71791677,  1.61756251])
>>> np.cos(x) + x[::-1]
array([ 1.,  2.,  3.,  4.])
```

Пример 2.17. Пусть необходимо решить следующее интегро-дифференциальное уравнение в квадрате [0-1] x [0-1]:

$$(\partial_x^2 + \partial_y^2)P + 5\left(\int_0^1 \int_0^1 cosh(P)dx\,dy\right)^2 = 0$$

при условии $P(x,1) = 1$ и $P = 0$ на других границах квадрата.

Это можно сделать, заменяя функцию P набором ее значений на некоторой сетке $P_{n,m} \approx P(nh, mh)$, с малым шагом h. Производные и интеграл можно заменить суммами, например

$$\partial_x^2 P(x,y) \approx (P(x+h,y) - 2P(x,y) + P(x-h,y))/h^2.$$

Тогда задача превращается в систему $N_x \times N_y$ нелинейных алгебраических уравнений, из которых можно найти значения функции во всех узлах сетки.

Из-за того, что число точек велико, стандартные методы поиска многомерных корней займут очень много времени. Решение можно найти специальными алгоритмами для больших систем, например, `krylov`, `broyden2` или `anderson`. Эти методы похожи на метод Ньютона, в которых вместо вычисления Якобиана (матрицы производных) находится ее приблизительное значение.

Решение можно найта при помощи алгоритма `newton_krylov`. Задаем сетку из 75×75 узлов и функцию для оптимизации `residual`. Обратите внимание, что значения на границах вычисляются иначе, чем внутри области. Решение находится методом итерации из начального нулевого приближения.

```
import numpy as np
from scipy.optimize import newton_krylov
from numpy import cosh, zeros_like, mgrid, zeros

# parameters
nx, ny = 75, 75
hx, hy = 1./(nx-1), 1./(ny-1)
```

```
P_left, P_right = 0, 0
P_top, P_bottom = 1, 0

def residual(P):
   d2x = zeros_like(P)
   d2y = zeros_like(P)

   d2x[1:-1] = (P[2:]    - 2*P[1:-1] + P[:-2]) / hx/hx
   d2x[0]    = (P[1]     - 2*P[0]    + P_left)/hx/hx
   d2x[-1]   = (P_right - 2*P[-1]    + P[-2])/hx/hx

   d2y[:,1:-1] = (P[:,2:] - 2*P[:,1:-1] + P[:,:-2])/hy/hy
   d2y[:,0]    = (P[:,1]  - 2*P[:,0]    + P_bottom)/hy/hy
   d2y[:,-1]   = (P_top   - 2*P[:,-1]   + P[:,-2])/hy/hy

   return d2x + d2y + 5*cosh(P).mean()**2

# solve
guess = zeros((nx, ny), float)
sol = newton_krylov(residual, guess, method='lgmres', verbose=1)
print 'Residual', abs(residual(sol)).max()
```

Строим график (см. гл. 2.6).

```
import matplotlib.pyplot as plt
x, y = mgrid[0:1:(nx*1j), 0:1:(ny*1j)]
plt.pcolor(x, y, sol)
plt.colorbar()
plt.show()
```

Алгебра и матрицы

Модуль линейной алгебры (scipy.linalg). Часть функций этого пакета импортирована из пакета numpy.linalg, однако в некоторых из них реализованы другие алгоритмы, а также добавлено много новых функций. В модуль вошли функции для решения матричных уравнений, поиска обратных матриц, решения задачи на собственные значения, разложения матриц, вычисления матричных функций через ряды Тейлора, создания матриц специального вида, и др.

Пример 2.18. Решаем систему уравнений:

$$\begin{cases} 1 \cdot x_1 + 2 \cdot x_2 = 5 \\ 3 \cdot x_1 + 4 \cdot x_2 = 6 \end{cases}$$

Задаем матрицу коэффициентов A и столбец значений b. Можно решать систему через поиск обратной матрицы `inv(A)`, но этот способ медленный и может содержать значительные ошибки (для матриц большого размера). Лучше использовать прямой способ `solv(A,b)` для решения системы.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5],[6]])
>>> b
array([[5],
       [6]])
>>> linalg.inv(A).dot(b) #slow
array([[-4. ],
       [ 4.5]]
>>> A.dot(linalg.inv(A).dot(b))-b #check
array([[  8.88178420e-16],
       [  2.66453526e-15]])
>>> np.linalg.solve(A,b) #fast
array([[-4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A,b))-b #check
array([[ 0.],
       [ 0.]])
```

Работа с разреженными матрицами (scipy.sparse). Поддерживает работу с разреженными матрицами различного вида: блочными, многодиагональными. Имеет семь классов разреженных матриц:

1. `csc_matrix`: сжатый по колонкам формат (Compressed Sparse Column format)
2. `csr_matrix`: сжатый по строкам формат (Compressed Sparse Row format)
3. `bsr_matrix`: блочный для разреженных по строкам (Block Sparse Row format)
4. `lil_matrix`: Список списков (List of Lists format)
5. `dok_matrix`: словарь ключей (Dictionary of Keys format)
6. `coo_matrix`: Координатный формат (COOrdinate format (aka IJV, triplet format))
7. `dia_matrix`: диагональный формат (DIAgonal format)

Для эффективного построения матрицы желательно использовать форматы `lil_matrix` или `dok_matrix`. Класс `lil_matrix` поддерживает основные операции (взятие срезов, индексирование), как массивы NumPy. Формат COO тоже может быть использован для эффективного создания матриц.

Для операций умножения или обращения матрицы сначала матрица должна быть конвертирована в матрицу формата CSC или CSR. Матрицы `lil_matrix` расположенны по сторкм, поэтому преобразование в

CSR эффективно, в отличие от преобразования в CSC. Все преобразования между форматами CSR, CSC, и COO имеют линейную по времени эффективность. Для разреженных матриц реализованы процедуры, позволяющие решать линейные уравнения прямыми и итерационными методами, находить собственные значения и собственные векторы, однако эффективность достигается только в случае, если матрица имеет строго определенный вид (например трехдиагональный) во всех других случаях эффективность может падать из-за временных задержек при поиске элементов таких матриц.

Пример 2.19. Задание диагональных матриц.

```
>>> import numpy as np
>>> from scipy.sparse import dia_matrix
>>> dia_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)

>>> data = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
>>> offsets = np.array([0, -1, 2])
>>> dia_matrix((data, offsets), shape=(4, 4)).toarray()
array([[1, 0, 3, 0],
       [1, 2, 0, 4],
       [0, 2, 3, 0],
       [0, 0, 3, 4]])
```

Пример 2.20. Решение системы уравнений с разреженными матрицами

Создаем блочную матрицу `lil_matrix` размером 1000x1000. Заполняем сто значений ее нулевой строки случайными значениями и дублируем их в первую строку со сдвигом, на диагонали ставим случайные значения. Конвертируем матрицу в CSR формат и решаем уравнение $Ax = b$ для столбца b, заполненного случайными числами:

```
import numpy as np
import scipy.sparse as sps
from scipy.sparse.linalg.dsolve import linsolve

rand = np.random.rand

mtx = sps.lil_matrix((1000, 1000), dtype=np.float64)
mtx[0, :100] = rand(100)
mtx[1, 100:200] = mtx[0, :100]
mtx.setdiag(rand(1000))

mtx = mtx.tocsr()
rhs = rand(1000)

x = linsolve.spsolve(mtx, rhs)
```

```
print 'rezidual:', np.linalg.norm(mtx * x - rhs)
```

Модуль специальных функций

Модуль для работы со специальными функциями (scipy.special). Большинство из функций, реализованных в данном модулей векторизованы (могут применятся к массивам). В модуль входят алгоритмы вычисления большого числа функций:

- эллиптические функции (Elliptic)
- Бесселя (Bessel), нули функций Бесселя, быстрые версии распространенных функций Бесселя, интегралы от функций Бесселя, производные функций Бесселя, сферические функции Бесселя, функции Бесселя—Рикати,
- Гамма функция и относящиеся к ней,
- функция ошибок и интегралы Френеля (Fresnel)
- функции Лежандра (Legendre),
- ортогональные полиномы (внимание! полиномы высоких порядков численно неустойчивы),
- Сферические волновые (Spheroidal Wave) функции,
- многие другие специальные функции.

Пример 2.21. Вычисление Гамма функции

```
>>> from scipy.special import gamma
>>> gamma(0.5)
1.7724538509055159
```

Модуль статистики

В отдельный модуль вынесены функции для работы со статистикой (scipy.stats). В этот модуль собраны функции, генерирующие статистические распределения, а также функции для статистической обработки массивов данных.

В модуль включены более 80 непрерывных распределений и более 10 дискретных распределений. Все непрерывные распределения относятся к классу `rv_continous`, для которого существует ряд методов, возвращающих основные параметры объекта. Аналогично дискретные распределения относятся к классу `rv_discrete`. Основные методы перечислены в таблице 2.4.

Статистические функции являются расширенной версией статистических функций, реализованных в пакете Numpy. Сюда входят алгебраи-

Таблица 2.1.: Методы случайных распределений.

Метод	**Описание**
`Rc.rvs(*args, **kwargs)`	Случайная величина данного типа
`Rc.pdf(x, *args, **kwds)`	Значение функции плотности вероятности в данной точке x
`Rc.logpdf(x, *args, **kwds)`	Логарифм функции плотности вероятности в данной точке x
`Rd.pmf(k, *args, **kwds)`	Значение функции вероятности в данной точке x
`Rd.logpmf(k, *args, **kwds)`	Логарифм функции вероятности в данной точке x
`Rc.cdf(x, *args, **kwds)`	Значение функции распределения в точке x
`Rd.moment(n, *args, **kwds)`	n-й нецентральный момент распределения
`Rc.median(*args, **kwds)`	Медиана распределения
`Rc.mean(*args, **kwds)`	Среднее значение
`Rc.std(*args, **kwds)`	Стандартное отклонение для распределения

ческие и геометрические средние, нахождение моментов распределений, получение частот и получение гистограм, проверка различных статистических критериев, и т.д.

Пример 2.22. Нормальное распределение. Получим набор из 5 случайных чисел с нормальным распределением и параметрами (0,3). Рассчитаем значение функции распределения (вероятность того, что случайная величина примет значение, меньшее или равное x) в точках из этого набора.

```
>>> from scipy.stats import norm
>>> a = norm.rvs(0, 3, size=5)
>>> a
array([-0.9647763 ,  4.04869472, -0.27537248,  7.89311625,
       -0.74648842])
>>> norm.cdf(a, 0, 3)
array([ 0.63055866,  0.74750746,  0.84134475,  0.90878878,
        0.95220965])
```

Пример 2.23. Сравнение распределений. Задаем выборки 1 и 2 из одного распределения (нормальное распределение с параметрами (5,10)) и выборку 3 с параметрами (8,10). Сравнение по методу Колмогорова–Смирнова показывает, что первые выборки похожи (соответствие 99%), а третье распределение отличается от первого (соответствие меньше 1%).

```
>>> rvs1 = stats.norm.rvs(loc=5, scale=10, size=500)
>>> rvs2 = stats.norm.rvs(loc=5, scale=10, size=500)
>>> rvs3 = stats.norm.rvs(loc=8, scale=10, size=500)

>>> stats.ks_2samp(rvs1, rvs2)
(0.025999999999999995, 0.99541195173064878)

>>> stats.ks_2samp(rvs1, rvs3)
(0.11399999999999999, 0.0027132103661283141)
```

Задания

2.4.1. Найдите минимум функции $sin(x*y)*cos(x*x)$ в квадрате 10x10 одним из методов пакета Scipy.optimize
Вычислите значения функции в узлах сетки с шагом 0.1 и найдите наименьшее из них
Сравните полученные результаты

2.4.2. Найдите значение интеграла $\int_0^\infty e^{-x^3+x}dx$

2.4.3. Подберите наилучший способ приближения функции $sin^2(x)$ на отрезке $[0,\pi]$ четырьмя функциями Гаусса

2.4.4. Задайте трехдиагональную квадратную матрицу A размера 100 со случайными значениями. Решите матричное уравнение A*X=B, где B –

единичная матрица.
Найдите обратную к матрице А при помощи функции из модуля `scipy.linalg`
Сравните результаты

2.5. Библиотека Matplotlib. Общая информация

Неотъемлемой частью научного процесса является визуализация данных. В Python существует несколько модулей, предназначенных для этой цели; наиболее используемым из них является Matplotlib.

Matplotlib — библиотека (набор модулей) Python для построения высококачественных 2D и 3D изображений и графиков. Matplotlib поддерживает несколько режимов работы, может быть использован интерактивно, поддерживает большинство распространенных растровых и векторных форматов изображений; графика Matplotlib легко встраивается в собственные приложения и на Интернет-страницы.

Библиотека Matplotlib была создана Дж. Хантером на основе MATLAB. Она построена на принципах ООП, однако, у нее есть процедурный интерфейс `pyplot`, напоминающий MATLAB. Использование этого интерфейса будет разобрано в гл. 2.6.

Установка Matplotlib происходит аналогично установке других модулей (см. гл. 1.6). Она отлично документирована — документация есть на сайте проекта `http://matplotlib.org`. Кроме документации по всем модулям библиотеки, на сайте есть галерея изображений, построенных с помощью Matplotlib; есть возможность скачать исходный код примера и изменить для своего графика.

Matplotlib имеет несколько зависимостей — при установке библиотеки необходимо обратить внимание на то, устаовлены ли зависимости. Они включают:

- Numpy — стандартная библиотека Python для работы с большими массивами данных; необходима для работы;
- pytz и python-dateutil — библиотеки работы с датами;
- IPython — интерактивная оболочка интерпретатора. Поддерживает историю команд, автодополнение, различные режимы работы; очень рекомендуется.

Задания

2.5.1. Установите Matplotlib на компьютер, если он еще не установлен.

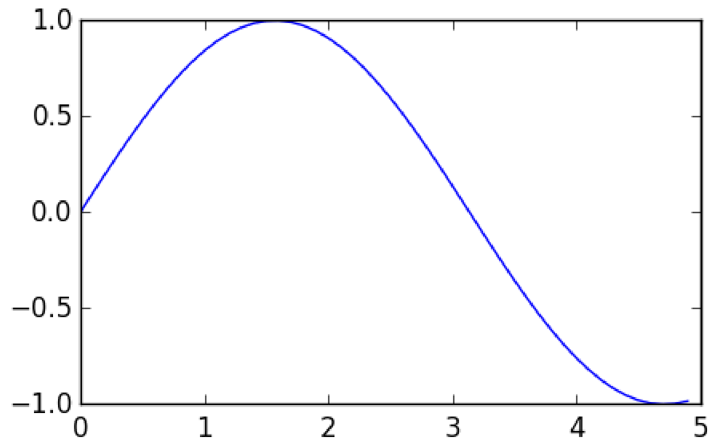


Рис. 2.1.: График $y = \sin(x)$.

2.6. Интерфейс Pyplot

Как уже было сказано, для построения простых графиков в Matplotlib существует интерфейс Pyplot. Импортируем его в нашу программу, чтобы начать использовать:

```
import matplotlib.pyplot as plt
```

Хотя мы можем импортировать `matplotlib.pyplot` под любым именем, `plt` используется в документации Matplotlib и поэтому является де-факто стандартом.

Девиз Matplotlib — «Делать простые вещи просто, а сложные — возможными». Простые графики в Matplotlib можно строить одной командой: `plt.plot`.

Пример 2.24. Рассмотрим пример:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0., 5., 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

Результат выполнения программы показан на рис. 2.1.

Разберем программу посрочно.

Первые две строки программы импортируют необходимые модули. Третья и четвертая строки создают данные для построения графика — независимая переменная x представляет собой массив значений от 0 до 5 (не включая 5) с шагом 0,1, созданный командой (numpy.arange); переменная y — массив, полученный поэлементным применением функции $\sin$ к

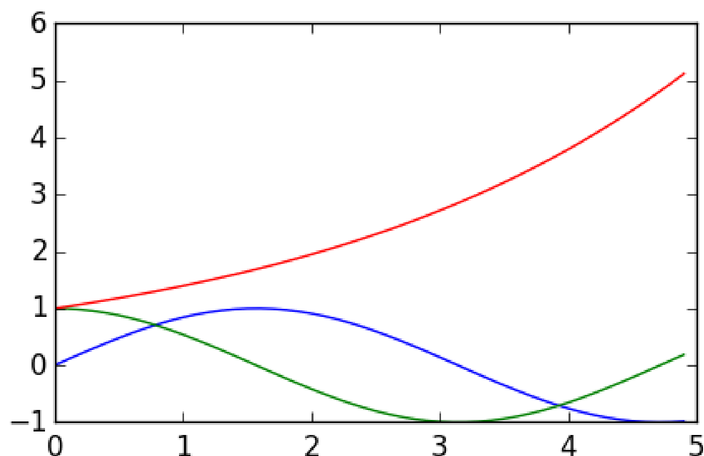


Рис. 2.2.: Графики $y = \sin(x)$, $y = \cos(x)$, $y = \exp(x/3)$.

массиву x. График строится строкой `plt.plot`, принимающей в качестве аргументов массивы x и y, и выводится на экран командой `plt.show`.

Несколько графиков в одном окне строятся с помощью нескольких команд `plt.plot` перед `plt.show`. При этом Matplotlib строит разные графики различными цветами (см. рис. 2.2; здесь и в дальнейшем первые две строки (импорт модулей) опускаются):

```
x = np.arange(0., 5., 0.1)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.plot(x, np.exp(x/3.))
plt.show()
```

Функция `plt.plot` принимает большое количество аргументов, способных изменить внешний вид графика. Большинство часто испоьзуемых аргументов описано в приложении C.1. Рассмотрим подробнее некоторые из них.

Для изменения цвета графика используется агрумент `color` (или `c`). Значением аргумента может быть либо название цвета по английски (например, `red`), либо положение цвета в RGB пространстве (три шестнадцатиричных числа от 00 до FF, каждое из которых показывает интенсивность базового цвета, например, `#0ADB25`). Стиль линии (или маркеров) определяется значением аргумента `linestyle` (или `ls`): `'-'` — сплошная линия, `'--'` — прерывистая линия, `'-.'` — тире-точка, `'s'` — квадратные маркеры и т.д. Полный перечень стилей линий и маркеров можно найти в документации Matplotlib. Цвет и стиль линии можно задать третьим неименованным аргументом `plt.plot`: он должен быть строкой, первый символ в которой — аббревиатура названия цвета (черный цвет обозначается `'k'`, остальные — начальной буквой названия), а остальные

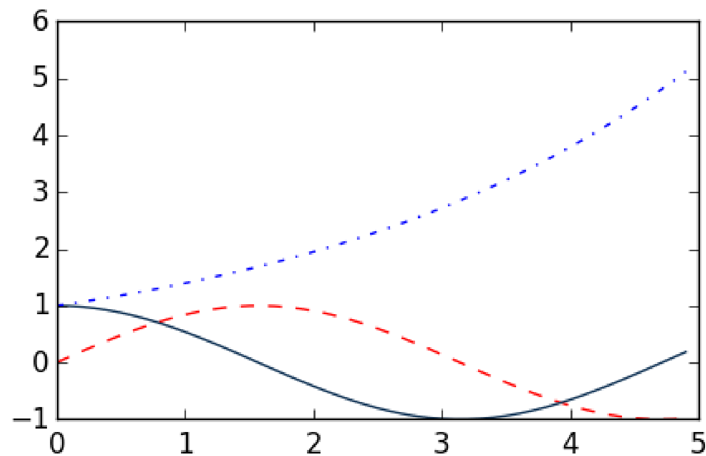


Рис. 2.3.: Выбор цвета и стиля графика.

символы — стиль линии.

Пример 2.25. Следующая программа демонстрирует выбор цвета и стиля линии.

```
x = np.arange(0., 5., 0.1)
plt.plot(x, np.sin(x), color='red', linestyle='--')
plt.plot(x, np.cos(x), c='#123456', ls='-')
plt.plot(x, np.exp(x/3.), 'b-.')
plt.show()
```

Результат работы программы представлен на рис. 2.3.

Кроме собственно построения графика, интерфейс pyplot имеет функции для основных настроек графика. Так, `plt.grid()` включает координатную сетку на графике; `plt.axis()` возвращает (при вызове без параметров) или изменяет (при вызове с параметрами `[xmin, xmax, ymin, ymax]`) пределы осей графика; `plt.xlabel()`, `plt.ylabel()` добавляют подписи к осям и т.д. Некоторые функции описаны в приложении C.1.

Для добавления легенды при построении графика к основным аргументам команды `plt.plot` необходимо добавить `label`. После этого на поле графика легенда добавляется командой `plt.legend`. Необязательный параметр `loc` определяет местоположение легенды на графике.

Пример 2.26. В легенде могут использоваться команды LaTeX:

```
x = np.arange(0., 5., 0.1)
plt.plot(x, np.sin(x), color='red', linestyle='--',
         label='$\sin(x)$')
plt.plot(x, np.exp(x/3.), 'b-.', label='$e^{x/3}$')
plt.legend(loc='best')
plt.show()
```

Полученный график представлен на рис. 2.4 слева. Заметьте, как работает значение `best`

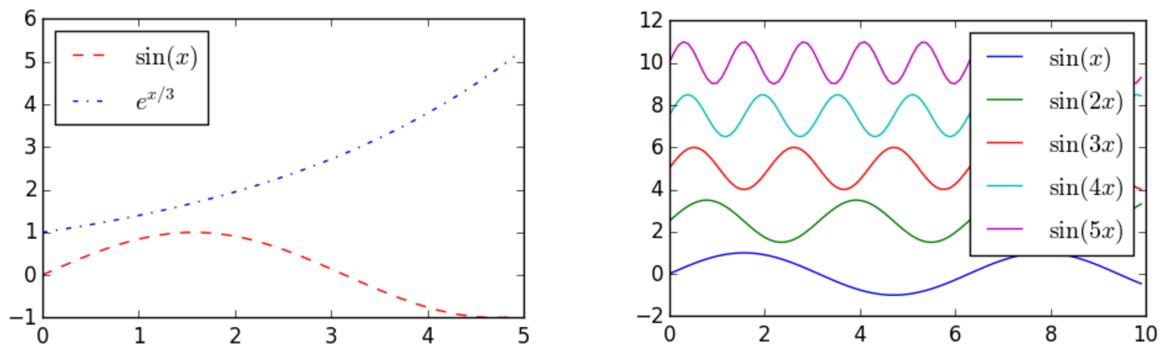


Рис. 2.4.: Графики с легендой.

параметра `loc`: Matplotlib самостоятельно пытается найти лучшее место для легенды так, чтобы она не закрывала линии графиков.

Задания

2.6.1. Напишите программу, строящую график на рис. 2.4 справа.

2.7. Объекты Matplotlib

Интерфейс `pyplot`, рассмотренный ранее, достаточно прост; Matplotlib, однако, предлагает еще один интерфейс, основанный на принципах ООП. Этот интерфейс чуть более сложен в использовании, но в то же время гораздо гибче `pyplot`; поэтому разработчики Matplotlib рекомендуют использовать именно его.

Рассмотрим, что происходит, когда мы строим график с помощью команды `plt.plot`:

- Создается экземпляр класса Figure, описывающий окно графика и его свойства, а также содержащий список всех его элементов;
- Создается связанный с Figure экземпляр класса Axes, представляющий собой описание поля графика, его осей и остальных свойств;
- Создаются экземпляры класса Line2D, включающие в себя данные, по которым строится график, его цвет и стили, и возвращается их список.

Каждый из указанных объектов имеет методы, изменяющие его свойства. Например, для изменения цвета линии мы можем использовать метод `set_color`, изменяющий цвет объекта `Line2D`:

```
x = np.arange(0., 5., 0.1)
l, = plt.plot(x, np.sin(x))
l.set_color('red')
```

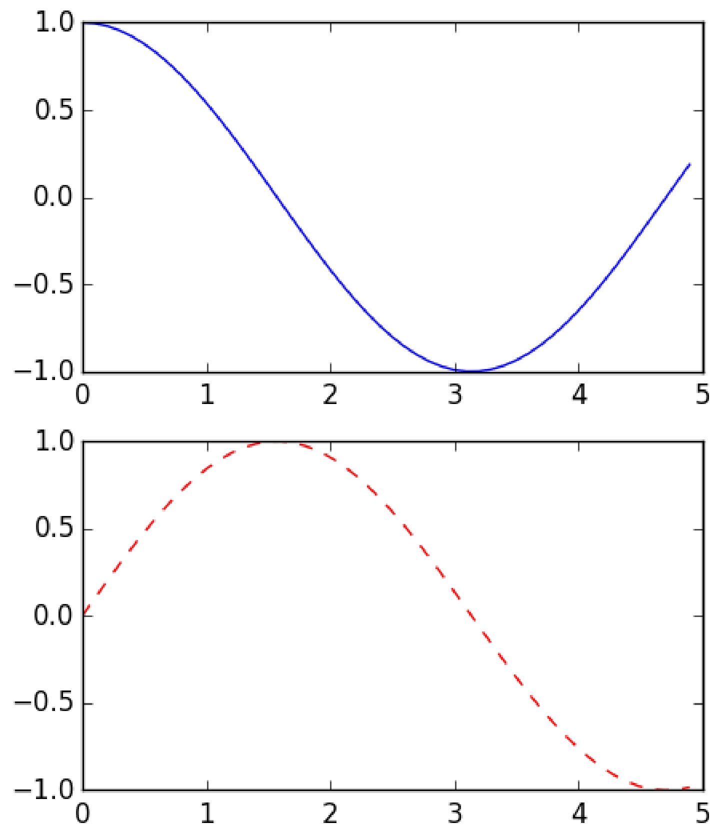


Рис. 2.5.: Построение графиков один под другим на различных осях.

`l,` — это кортеж из одного элемента (экземпляра класса Line2D), возвращаемый командой `plt.plot`. Когда последовательности переменных присваивается последовательность значений, каждой переменной присваивается соответствующее ей значение. Таким образом, в `l` мы получаем первый элемент кортежа, возвращенного `plt.plot`.

Бывает так, что на одном рисунке необходимо построить несколько графиков, занимающих разные оси. Для этого надо связать несколько экземпляров класса Axes с одним Figure; при этом используется метод `add_subplots` класса Figure. Этот метод принимает аргументы, описывающие геометрию рисунка — количество рядов и колонок графиков, а также номер графика на рисунке.

Пример 2.27. Построим два графика на одном рисунке один над другим:

```
fig = plt.figure()
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)
x = np.arange(0., 5., 0.1)
ax1.plot(x, np.cos(x), 'b-')
ax2.plot(x, np.sin(x), 'r--')
plt.show()
```

Разберем пример построчно. Первая строка возвращает экземпляр класса Figure, к которому мы можем добавить один или несколько Axes. Вторая и третья строки добавляют к Figure два объекта Axes, сгруппированные в 2 ряда и 1 колонку, причем переменная ax1 связана с первым таким объектом, а ax2 — со вторым. Пятая и шестая строки вызывают метод `plot` объекта `Axes`, строящий графики. Заметьте, что мы самостоятельно выбираем цвет и стиль линии. Результат выполнения программы показан на рис. 2.5.

Задания

2.7.1. Постройте график, показанный на рис. 2.6. Используйте для этого команды из приложения C.2; документация к ним доступна в Интернете и интерактивной справке.

2.8. Анимация в Matplotlib

Начиная с версии 1.1, Matplotlib поддерживает создание анимированных графиков и их экспорт в видео-файл. Построение анимаций происходит на базе классов `TimedAnimation` и `FuncAnimation`, опеделеных в модуле `matplotlib.animation`.

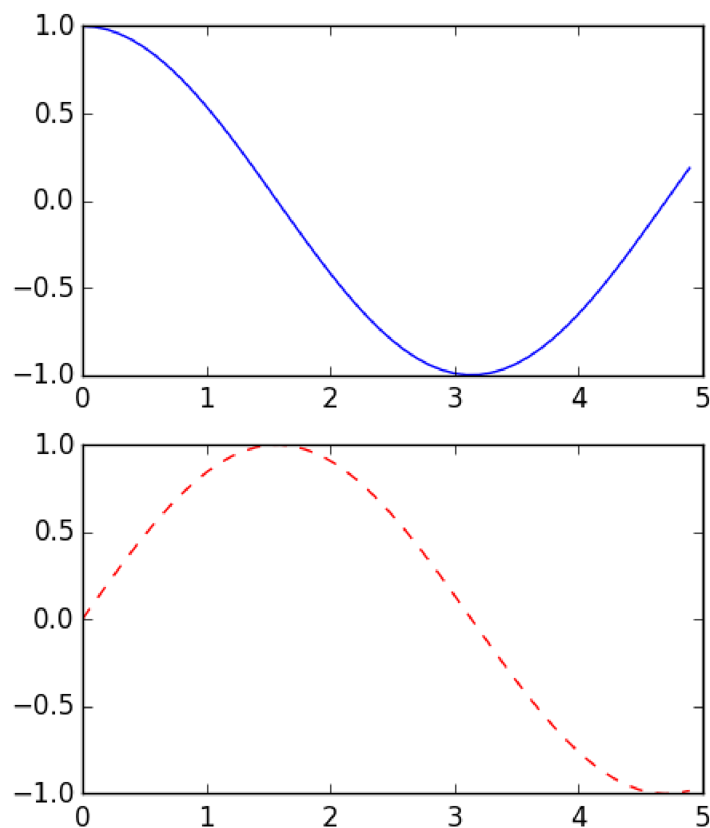


Рис. 2.6.: К заданию 2.7.1.

Мы будем рассматривать построение анимаций с помощью класса FuncAnimation. Экземпляр этого класса принимает при создании анимации функцию, имеющую аргументов номер шага по времени, и возвращающую список объектов класса Line2D, которые затем помещаются на поле графика.

Объект FuncAnimation при создании принимает:

- объект Figure, на котором будет строиться анимация;
- функцию animate, изменяющую линию;
- количество кадров frames (может быть также итератором);
- (по желанию) функцию ini_func, инициализирующую график (выполняется в начале построения анимации либо в начале построения каждого кадра, если blit=True);
- interval — интервал между кадрами в миллисекундах;
- логический параметр blit, включающий перерисовку только изменяющихся частей графика.

Таким образом, можно сформулировать следующий алгоритм построения анимации:

1. вызвать конструктор plt.figure, создающий объект Figure;
2. создать изменяемый объект (обычно Line2D, создаваемый командой plt.plot);
3. написать функцию, изменяющую объект;
4. создать анимацию.

Работу этого алгоритма рассмотрим на следующем примере, реализующем бегущую волну.

Пример 2.28. Этот пример иллюстрирует построение бегущей синусоиды. Для того, чтобы синусоида двигалась, мы сдвигаем ее на 0,1 влево за каждый временной шаг.

```
import matplotlib.animation as animation

fig, ax = plt.subplots()
x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))

def init():
    line.set_ydata(np.ma.array(x, mask=True))
    return line,

def animate(i):
    line.set_ydata(np.sin(x+i/10.0))
    return line,

ani = animation.FuncAnimation(fig, animate, frames=200,
    init_func=init, interval=25, blit=True)
plt.show()
```

Задания

2.8.1. Создайте анимацию горизонтальной линии, движущейся вверх и вниз.

2.8.2. Создайте анимацию падающего шара.

3. Решение физических задач

3.1. Применение компьютера для решения физических задач

Начать разговор о применении ЭВМ для решения физических задач следует с чёткой постановки вопроса. Обычно под задачей, возникающей в физике, подразумевается некоторая ситуация и желание получить ответ на вопрос: «Что будет происходить дальше?» (прямая задача) или «Какие условия вызвали появление сложившейся ситуации?» (обратная задача). Решение физической задачи строится на *начальных данных* и *законах*, определяющих связь между параметрами системы. Таким образом решение состоит из:

1. описания ситуации и формулировки вопроса, на который нужно получить ответ;
2. выявления основных законов, управляющих процессами в изучаемых системах;
3. отбора необходимых данных;
4. формулировки математической задачи;
5. решения математической задачи (получения числа или набора чисел);
6. анализа результата и формулировки ответа на поставленный вопрос.

Сразу заметим, что пункты 1, 3 и 6 часто исключаются из работы школьника и студента, когда они решают «чужие» задачи, но совершенно необходимы при самостоятельной научной деятельности. Стоит обратить внимание на отличие физической задачи от задачи математической, которая включает только в пункты 4 и 5 представленного списка. Для решения физической задачи необходимо провести «предварительную» работу по пунктам 1–3 и заключительную работу по пункту 6, основанную целиком на всестороннем индивидуальном и неформальном анализе физической проблемы.

Пунктам 1–3 мы не будем уделять много внимания в данном пособии, т.к. студенты должны хорошо разобраться в этих вопросах при изучении специальных физических дисциплин. Тем более, что компьютер пока не в состоянии оказать помощь учёному в этих вопросах. Тем не менее ниже подробно разобран пример, показывающий полный алгоритм рассуждений при решении очень известной задачи о попадании в цель снарядом из пушки.

Анализ физической проблемы

Ниже приведен пример анализа физической задачи и этапов подготовки ее для решения при помощи ЭВМ.

Пример 3.1. *Описание ситуации*: имеется пушка, из ствола которой вылетает снаряд со скоростью v_0. Требуется определить, как направить дуло пушки, чтобы снаряд поразил цель, находящуюся на определённом расстоянии от нее на том же уровне по горизонтали.

Выявление основных законов и *отбор необходимых данных*

Из курса физики средней школы хорошо известно, что, выбрав систему координат с началом в точке, где расположена пушка, и одной из осей, направленной на цель, можно свести задачу к системе двух уравнений, для решения которых необходимо знать скорость вылета снаряда v_0, расстояние до цели L и ускорение свободного падения g. Данная ситуация справедлива, если маломощная пушка стреляет маленькими тяжелыми ядрами, что вполне подходило для орудий до XIX века.

При увеличении скорости снаряда и прицельного расстояния оказалось необходимым дополнительно учесть силу сопротивления воздуха, которая снижает дальность полета. Сила сопротивления воздуха зависит от формы снаряда и скорости его полета:

$$F_c = C\frac{\rho v^2}{2},$$

где ρ — плотность воздуха, v — скорость полета снаряда, C — аэродинамический коэффициент. Таким образом, для учета сопротивления воздуха необходимо дополнительно знать аэродинамический коэффициент и плотность воздуха.

Стоит отметить, что плотность воздуха зависит от высоты:

$$\rho(h) = \rho_0 \cdot \exp\left(-\frac{\mu g h}{kT}\right),$$

где μ — молярная масса воздуха, g — ускорение свободного падения, k — постоянная Больцмана, T — средняя температура атмосферы, т.е. при баллистических траекториях сила сопротивления может оказаться различной на разной высоте. Поправки на сопротивление воздуха устраивали оружейников до конца XIX века. Однако появление «суперпушек» выявило следующий недостаток.

При больших скоростях полета проявляется сила Кориолиса, отклоняющая снаряд в сторону от направления выстрела: $\vec{F_k} = 2m[\vec{v} \times \vec{\omega}]$, где ω — угловая скорость вращения Земли. Таким образом, задача полета снаряда превращается в трехмерную задачу с изменяющимися силами. При этом дополнительно требуется знать местоположение пушки и цели на Земном шаре (широту и направление выстрела).

С появлением неуправляемых реактивных снарядов произошло дальнейшее усложнение методики прогнозирования попадания снаряда в цель. Для очень «высоких» траекторий необходимо учитывать изменение ускорения свободного падения с высотой, кривизну земной поверхности и т.д.

Мы остановимся на приближении нереактивного снаряда, летящего в атмосфере с постоянной плотностью. Задача о попадании снаряда в цель сводится к изучению движения под действием трех сил:

$$m\frac{d\vec{v}}{dt} = m\vec{g} - C\frac{\rho v\vec{v}}{2} + 2m[\vec{v} \times \vec{\omega}]. \tag{3.1}$$

Для решения задачи необходимо знать m, C, ρ, v_0, L, широту расположения пушки ϕ и направление от пушки на цель.

Математическая формулировка задачи

Математическая формулировка в случае простых задач обычно не вызывает затруднений, но в случае больших задач может иметь сложности. Математическая формулировка включает в себя:

1. исходные данные;
2. требуемый результат;
3. метод получения результата из исходных данных.

В качестве исходных данных берутся все физические (т.е. имеющие некоторый физический смысл) величины, присутствующие в исходной задаче. В качестве результата необходимо выбрать величину или набор величин, из которых можно получить ответ на требуемый вопрос. Стоит отметить, что желательно рассмотреть все возможные методы получения результата: попытаться дать аналитическую и геометрическую формулировку задачи, а также представить максимальное число методов получения результата из исходных данных. Если вычисления содержат цепочку промежуточных результатов, каждый из которых получается своим методом, то задачу необходимо разбить на простые подзадачи и сформулировать положения 1–3 для каждой из них. На этом этапе нужно особое внимание уделить единицам измерения и возможному их упрощению.

Пример 3.2. Для примера с пушкой, разобранного выше, математическая задача появляется после введения системы координат. Для упрощения вычислений ось Ox нужно направить на север (по меридиану), Oy — на зпад (по параллели), Oz — вверх (от центра земли). Тогда проекции угловой скорости вращения земли будут равны $(\omega\cos(\phi), 0, \omega\sin(\phi))$. Уравнение 3.1 превратится в систему обыкновенных дифференциальных уравнений:

$$\begin{cases} m\dfrac{dv_x}{dt} = -C\dfrac{\rho v v_x}{2} + 2mv_y\omega\sin(\phi), \\ m\dfrac{dv_y}{dt} = -C\dfrac{\rho v v_y}{2} + 2m(v_z\omega\sin(\phi) - v_x\omega\cos(\phi)), \\ m\dfrac{dv_z}{dt} = -mg - C\dfrac{\rho v v_z}{2} + 2mv_y\omega\cos(\phi). \end{cases} \tag{3.2}$$

Для этой системы уравнений необходимо решить краевую задачу, т.е. найти решение, удовлетворяющее условиям $x(0) = 0$, $y(0) = 0$, $z(0) = 0$; $x(t) = X_{target}$, $y(t) = Y_{target}$, $z(t) = Z_{target} = 0$, $\sqrt{v_x(0)^2 + v_y(0)^2 + v_z(0)^2} = v_0$, где t — некоторое время.

Результатом будут являться значения проекций скорости в начальный момент времени $(v_x(0), v_y(0), v_z(0))$, которые определяют направление выстрела. Из простых практических соображений можно заключить, что данная задача может быть решена хотя бы методом угадывания (перелет – недолет – попадение), который называют методом стрельбы. Описание алгоритма этого метода приведено далее.

Решение математической задачи

Решение математической задачи при помощи ЭВМ разделяется на ряд действий, без которых невозможно получение надежного результата:

1. выбор и обоснование метода решения;
2. построение алгоритма;
3. написание программы (реализация алгоритма);
4. тестирование программы на 2–3 примерах с известным ответом;
5. получение ответа на поставленную задачу.

При выборе и обосновании метода решения особое внимание нужно уделить поиску *всех* возможных методов решения и выбору среди них оптимального. Критериями оптимальности могут являться: быстрота работы программы, точность результата, простота реализации, возможность быстрой доработки или переноса на другую платформу и т.д. Обычно в качестве критерия выбирается *время решения задачи* при *обеспечении нужной точности*. Очень важно понимать, что время решения задачи — это не только время работы программы, но и время, затраченное на проработку алгоритма, написание и отладку программы, в общем, все время работы по пунктам 1–5. Для неопытных программистов это время может на порядки превосходить время проведения вычислений, поэтому во всем мире переходят на системы, позволяющие быстро генерировать код программы (пусть и более медленный) с использованием вычислительных пакетов MATLAB, MathCAD и.т.п., и языки программирования высокого уровня.

Пример 3.3. Решение примера с выстрелом из орудия целесообразно проводить методом стрельбы. Он состоит из прицеливания (выбора направления выстрела), выстрела (вычисления траектории полета снаряда для заданных начальных условий) и корректировки (изменения направления выстрела в зависимости от точки падения снаряда). С точки зрения алгоритмов данная процедура является задачей минимизации функции расстояния от точки падения снаряда до цели $D(a, b) \to \min$. Если снаряд может попасть в цель при заданных условиях, то минимум достигается, когда расстояние равно 0. Функция D зависит от двух переменных, определяющих в пространстве направление прицеливания. Таким образом необходимо решить две задачи: иметь способ вычисления функции D для любого направления пушки; уметь находить минимум функции двух переменных. Детали реализации алгоритмов рассмотрены ниже.

На этапе построения алгоритма программы необходимо проработать следующие вопросы:

- разделение задачи на небольшие части, имеющие стандартные алгоритмы решения;
- составление блок схемы (обычно достаточно указания основных блоков и данных, которыми они обмениваются);
- определение параметров, регулирующих выполнение программы (шаг

сетки, точности вычисления разных параметров и т.п.);
- оценка планируемого быстродействия и определение путей его повышения;
- оценка планируемой точности и определение путей ее повышения;
- предполагаемые случаи неправильной работы (недопустимые значения параметров или входных данных).

Пример 3.4. Распишем алгоритм для вычисления функции расстояния $D(a, b)$ для рассматриваемого примера.

– В качестве параметров, задающих направление скорости, выберем углы в полярной системе координат: $0 < a < \pi/2$ – угол с осью Oz, $0 \leq b \leq 2\pi$ – угол проекции скорости на плоскость xOy с осью Ox. Тогда проекции начальной скорости равны $(v_0 \sin(a) \cos(b), v_0 \sin(a) \sin(b), v_0 \cos(a))$.

– Запишем систему дифференциальных уравнений второго порядка как систему уравнений первого порядка по переменным координаты и скорости:

$$\begin{cases} \dfrac{dv_x}{dt} = -C\dfrac{\rho v v_x}{2m} + 2v_y\omega\sin(\phi), \\ \dfrac{dv_y}{dt} = -C\dfrac{\rho v v_y}{2m} + 2(v_z\omega\sin(\phi) - v_x\omega cos(\phi)), \\ \dfrac{dv_z}{dt} = -g - C\dfrac{\rho v v_z}{2m} + 2v_y\omega\cos(\phi), \\ \dfrac{dx}{dt} = v_x, \\ \dfrac{dy}{dt} = v_y, \\ \dfrac{dz}{dt} = v_z. \end{cases} \tag{3.3}$$

Перепишем эту систему в векторном виде:

$$\frac{d}{dt}\begin{bmatrix} v_x \\ v_y \\ v_z \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -C\dfrac{\rho v}{2m} \cdot v_x + 2\omega\sin(\phi) \cdot v_y \\ -2\omega\cos(\phi) \cdot v_x - C\dfrac{\rho v}{2m} \cdot v_y + 2\omega\sin(\phi) \cdot v_z \\ 2\omega * cos(\phi) \cdot v_y - C\dfrac{\rho v}{2m} \cdot v_z - g \\ v_x \\ v_y \\ v_z \end{bmatrix}. \tag{3.4}$$

Такая запись является общепринятой и используется в большинстве алгоритмов решения дифференциальных уравнений, т.к. позволяет быстро и формально вычислять правую часть уравнения в любой точке траектории. Решение задачи Коши (интегрирование дифференциальных уравнений при известных начальных условиях) для системы первого порядка проводим стандартными методами, реализованными во множестве библиотек. Можно использовать один из подходов, описанных в разделе 3.2. При этом точность интегрирования оценивается при сравнении результатов, полученных с разным шагом.

– Условием остановки вычислений является падение снаряда на землю: $z(\tau) \leq 0$. Координаты $x(\tau)$ и $y(\tau)$ в этот момент позволяют найти искомое расстояние: $D(a,b) = \sqrt{(x(\tau) - X_{target})^2 + (y(\tau) - Y_{target})^2}$

Задача поиска минимума функции $D(a,b)$ — это стандартная задача, которая может быть решена посредством использования библиотечных функций или путем реализации одного из известных численных методов. Стоит отметить, что минимум может быть не равен 0 (скорости снаряда не достаточно для преодоления расстояния до цели), а также, что решений может быть одно или два.

Для написания программы обычно выбирают язык или программный пакет, реализация алгоритма в котором производится наиболее быстро (с учетом навыка исполнителя).

Обязательным условием решения является *тестирование* полученной программы. Тестирование программы производится для оценки

- правильности работы программы;
- точности работы алгоритма;
- времени работы программы (если задачу планируется усложнять).

Для простейшего тестирования необходимо подобрать 2–3 тестовых примера (с известным результатом) и проверить правильность счета. При необходимости таким же способом проверяют работоспособность отдельных частей программы. Методы тестирования и методы анализа результатов во многом схожи, только при тестировании результат проверяется на соответствие принципам математики, а при анализе результатов исследуется физический смысл полученных данных.

Пример 3.5. В описанном примере необходимо проверить правильность работы в известных частных случаях:

- выстрел вблизи экватора (сила Кориолиса равна 0),
- отсутствие воздуха (сила сопротивления равна 0).

Получение результатов, совпадающих с известными из теории, может являться подтверждением отсутствия грубых ошибок в алгоритме.

Анализ результатов

Анализ результатов представляет собой отдельную проблему, часто сравнимую по сложности с решением самой задачи. Особенно отчетливо это проявляется, когда в результате решения математической задачи появляется большое количество данных: поле скоростей движения частиц, поле температур, расположение многих тел в пространстве и даже обычная функциональная зависимость. Анализ этих данных требует дополнительных методик обработки информации, схожих с методами обработки натурных экспериментов. Наиболее распространены методы построения графиков и сечений, визуализации векторных и скалярных полей и т.д.

Пример 3.6. Анализ результата в разобранном примере заключается прежде всего в

проверке предположений теории, которые были сделаны при решении (пренебрежение изменением плотности воздуха с высотой, плоская поверхность земли и т.д.). После этого можно давать прогнозы о попадании снарядов в цель.

3.2. Задачи на движение отдельных тел

Задача о движении конечной системы тел сводится к системе дифференциальных уравнений, основной переменной в которых выступает время, а функциями являются координаты и скорости частей системы. Пример получения системы дифференциальных уравнений при рассмотрении физической задачи (движении снаряда при выстреле из пушки) был рассмотрен в разделе 3.1. В данном разделе будут более подробно разобраны особенности решения математической задачи с помощью ЭВМ.

Решение обыкновенных дифференциальных уравнений

Обыкновенные дифференциальные уравнения (ОДУ) или их системы записывают в виде:

$$\frac{d}{dt}x_i(t) = f_i(x_1, x_2, ...x_n, t) \tag{3.5}$$

для всех $i = 0...n$ или, в векторной записи:

$$\frac{d}{dt}\vec{x}(t) = \vec{f}(\vec{x}, t). \tag{3.6}$$

Уравнения высших порядков легко переписываются в таком виде после замены переменных. Результат решения системы ДУ может быть получен, если дополнительно заданы n условий вида $x_i(t_i) = C_i$.

В соответствии с заданными дополнительными условиями различают два типа задач с разным смыслом: задачу Коши и краевую задачу. Для задачи Коши все дополнительные условия заданы для одного состояния системы, для одной точки t, поэтому ее физическим смыслом является ответ на вопрос: «Как будет двигаться известная система в будущем (в прошлом)?» Для краевой задачи дополнительные условия заданы для разных состояний системы, разных точек t_i. Ее решение дает ответ на вопрос: «Как привести систему в нужное состояние?» Обычно решение ОДУ ищется через решение задачи Коши, т.к. она является более простой.

Решение задачи Коши

Задача Коши для ОДУ решается «шаговыми» методами — последовательным восстановлением функции $\vec{x}(t)$ по ее значению в начальной

точке $\vec{x}(0)$ и известной производной (функции $\vec{f}(\vec{x},t)$). Простейшим шаговым методом является метод Эйлера: $\vec{x}_{m+1} = \vec{x}_m + dt \cdot \vec{f}(\vec{x}_m, t)$, где dt — шаг по времени. Его схема приведена на рисунке 3.1a. В методе Эйлера восстанавливается ломаная линия. Ошибка решения метода Эйлера на каждом шаге порядка dt, поэтому решение методом Эйлера довольно быстро уходит от истинной функции. Можно повысить точность, уточняя направление шага, например, используя производную не в той точке, из которой мы делаем шаг, а в точке $dt/2$ (модифицированный метод Эйлера): $\vec{k}_1 = \vec{f}(\vec{x}_m, t)$; $\vec{k}_2 = \vec{f}(\vec{x}_m + \vec{k}_1 \cdot dt/2, t + dt/2)$; $\vec{x}_{m+1} = \vec{x}_m + dt \cdot \vec{k}_2$. Его схема приведена на рисунке 3.1b.

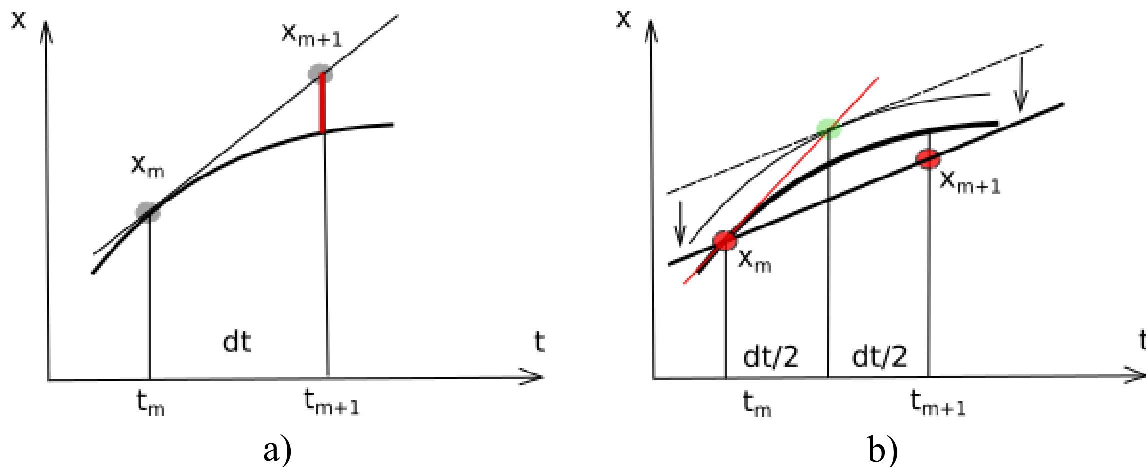


Рис. 3.1.: Один шаг при решении задачи Коши. a) - метод Эйлера, b) - модифицированный метод Эйлера.

Все «шаговые» методы носят названия методов Рунге–Кутты. Методы Рунге–Кутты третьего и высших порядков строятся по схожим схемам. Например схема метода четвёртого порядка задаётся уравнением:

$$\vec{x}_{m+1} = \vec{x}_m + dt \cdot (\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)/6, \tag{3.7}$$

где точное направление определяется линейной комбинацией правых частей в разных точках:

$$\begin{aligned}
\vec{k}_1 &= \vec{f}(\vec{x}_m, t), \\
\vec{k}_2 &= \vec{f}(\vec{x}_m + \vec{k}_1 \cdot dt/2, t + dt/2), \\
\vec{k}_3 &= \vec{f}(\vec{x}_m + \vec{k}_2 \cdot dt/2, t + dt/2), \\
\vec{k}_4 &= \vec{f}(\vec{x}_m + \vec{k}_3 \cdot dt, t + dt).
\end{aligned} \tag{3.8}$$

Точность решения ДУ

Для оценки точности решения необходимо учесть ошибки дискретизации (замены истинной гладкой функции на ломаную), которые пропорциональны dt или dt^2, ошибки округления (накопление ошибок при выполнении операций), которые пропорциональны dt^{-1}, а также устойчивость самого уравнения.

Уравнение может быть устойчивым (ошибки при решении не увеличиваются) или неустойчивым (все ошибки увеличиваются, часто экспоненциально). Для устойчивого уравнения полная ошибка не превосходит суммы ошибок, допущенных на каждом шаге (с точностью до постоянного множителя): $err < \alpha(e_1 + e_2 + e_3 + ...)$. Для неустойчивого уравнения полная ошибка больше суммы ошибок, допущенных на каждом шаге: $err > \alpha(e_1 + e_2 + e_3 + ...)$. Часто ошибка решения для неустойчивых уравнений неограниченно возрастает при продолжении решения. Для неустойчивых уравнений требуется тщательно выбирать метод решения и анализировать сходимость результата. Примеры поведения численного решения для устойчивых и неустойчивых уравнений приведены на рисунке 3.2.

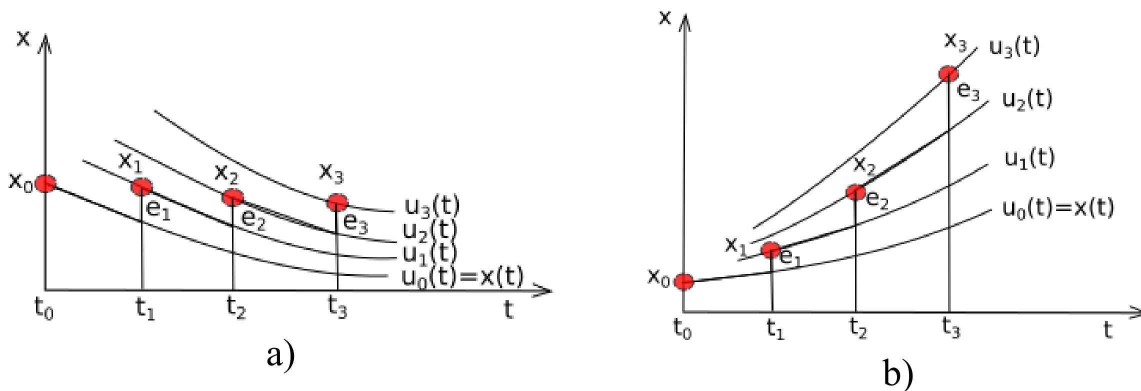


Рис. 3.2.: Результат применения шагового метода для решения a – устойчивого и b – неустойчивого дифференциального уравнения.

Решение краевой задачи

В случае краевой задачи из n условий, необходимых для однозначного решения задачи, на левом конце временного интервала заданы только часть (m штук):

$$L_k(\vec{x}(t_0)) = 0, k = 1...m,$$

остальные $n - m$ условий заданы на правом конце интервала:

$$R_k(\vec{x}(t_l)) = 0, k = 1...n - m.$$

Для решения краевых задач ОДУ используется метод стрельбы или метод релаксации.

Метод стрельбы

Метод стрельбы заключается в подборе недостающих параметров на левом конце интервала так, чтобы на правом конце интервала (после решения ОДУ) получилось нужное значение. Схема метода стрельбы проста:

- введем $n-m$ параметров $\alpha_1, \ldots, \alpha_{n-m}$ так, чтобы получить полный набор начальных условий на левом конце $\vec{x}(\alpha_1, \ldots, \alpha_{m-n}, t_0)$, удовлетворяющих ограничениям $L_k(\vec{x}(t_0)) = 0, k = 1...m$;
- стреляем, т.е. решаем задачу Коши для известных начальных значений $\vec{x}(\alpha_1 \ldots \alpha_{m-n}, t_0)$ и получаем значения на правом конце интервала $\vec{x}(\alpha_1 \ldots \alpha_{m-n}, t_l)$;
- проверяем условия на правом конце интервала $R_k(\vec{x}(\alpha_1 \ldots \alpha_{m-n}, t_l)) = 0, k = 1...n-m$. Если условия выполнены, то решение найдено.
- При необходимости корректируем параметры $\alpha_1 \ldots \alpha_{n-m}$, например, методом Ньютона:

$$\frac{\partial R_k}{\partial \alpha_i} = \frac{R_k(\vec{x}(\alpha_1 \ldots \alpha_i + \epsilon \ldots \alpha_{m-n}, t_l)) - R_k(\vec{x}(\alpha_1 \ldots \alpha_i \ldots \alpha_{m-n}, t_l))}{\epsilon};$$

$$\alpha_i = \alpha_i - \frac{R_k}{\partial R_k / \partial \alpha_i}$$

Таким образом, на каждую итерацию необходимо $n-m+1$ решений задачи Коши.

Несколько замечаний по методу стрельбы:

- Если больше условий задано на правом конце интервала ($m < n-m$), задачу Коши решаем в противоположную сторону (если это возможно).
- Возможны ситуации с неравнозначностью движения по траектории в разные стороны, например, для уравнения теплопроводности или уравнения, в решение которого входит возрастающая экспонента.
- Возможны ситуации, когда сшивать решения нужно в середине интервала, например, при решении уравнения Шредингера.

Релаксационные методы

Релаксационные методы основаны на формировании системы алгебраических уравнений для поиска значений функции $\vec{x}(t_k)$ в узлах некоторой *сетки*. В простейшем случае замены производной на отношение, система

уравнений, полученная из уравнений 3.6, имеет вид:

$$\frac{\vec{x}(t_{k+1}) - \vec{x}(t_k)}{\Delta t} - \frac{\vec{f}(x(t_{k+1})) - \vec{f}(x(t_k))}{2} = 0, k = 0...l, \tag{3.9}$$

$$L_k(\vec{x}(t_0)) = 0, k = 1...m; \tag{3.10}$$

$$R_k(\vec{x}(t_l)) = 0, k = 1...n - m, \tag{3.11}$$

Остается решить полученную систему алгебраических уравнений. В случае линейных систем получается линейная система уравнений, которая решается точно. Для нелинейных систем широко используются итерационные методы решения.

Примеры и задания

Маятники. Колебательные процессы появляются при наличии в системе положения устойчивого равновесия. Механическим примером таких систем являются маятники - тела, подвешенные в поле силы тяжести. Уравнение движения такой системы в простейшем случае имеет вид:

$$J\frac{d^2\phi}{dt^2} = -M(\phi),$$

где J - момент инерции системы, M - момент сил, возвращающих систему в положение равновесия.

3.2.1. **Математический маятник.** В случае математического маятника длины l

$$J = m \cdot l^2, M = mgl \cdot sin(\phi),$$

что дает уравнение движения

$$\frac{d^2\phi}{dt^2} + \frac{g}{l}sin(\phi) = 0.$$

Запишем уравнение в стандартном виде:

$$\frac{d}{dt}\begin{bmatrix}\phi\\ \dot{\phi}\end{bmatrix} = \begin{bmatrix}\dot{\phi}\\ -\frac{g}{l}sin(\phi)\end{bmatrix}. \tag{3.12}$$

Для его решения воспользуемся функцией `odeint()`, расположенной в модуле `scipy.integrate`.

- Задаем функцию, возвращающую столбец правых частей системы уравнений:

```
def force_math(f, t):
    return f[1], -g/L * sin(f[0])
```

Используется стандартный вид функции, принимающий обобщенные координаты точки (в данном случае значение угла и его первой производной) и время, и возвращающий значения производных от каждой компоненты вектора.

- Задаем моменты времени, для которых вычисляется положения маятника:

```
# Массив моментов времени:
N = 10000
t = np.linspace(0, 50, N)
```

- Решаем систему, получаем доступ к массиву координат, находим декартовы координаты маятника и строим графики:

```
# Начальные условия
R0 = np.array([np.pi*0.99, 0.])

R = integr.odeint(force_math, R0, t)

phi, phi_dot = R.T
x = L * np.sin(phi)
y = -L * np.cos(phi)

plt.plot(t, x)
plt.plot(t, y)
plt.show()
```

Задания:

1. Соберите программу, описывающую движение математического маятника.
2. Постройте траектории движения маятника при разных углах начального отклонения (от 0 до $\pi/2$). Опишите результат.
3. Смоделируйте движение маятника для длительного времени (несколько суток). проверьте закон сохранения энергии для полученных данных. Сделайте выводы.
4. * Рассмотрите движение маятника, не являющееся плоским (начальная скорость не лежит в вертикальной плоскости). Составьте программу, вычисляющую траектории движения маятника при различных начальных условиях. Изобразите траектории, получающиеся при таком движении.
5. * Учтите при движении маятника силу Кориолиса. Удается ли наблюдать опыт Фуко?

3.2.2. **Маятник Капицы.** Рассмотрим математический маятник, точка подвеса которого совершает гармонические колебания по гармоническому

закону в вертикальной плоскости:

$$y_o(t) = a\cos(\gamma t).$$

Уравнение движения такого маятника имеет вид

$$\frac{d^2\phi}{dt^2} + (\frac{g}{l} + \frac{a\gamma^2}{l}cos(\gamma t)) \cdot sin(\phi) = 0$$

Для решения уравнения зададим новую функцию, возвращающую столбец правых частей интегрального уравнения:

```
def force_kapitsa(f, t):
    return f[1], -(g/L + a/L*gamma**2*cos(gamma*t))*sin(f[0])
```

Решение проводим по алгоритму, использованному для обычного математического маятника и описанному выше.

Задания:

1. Соберите программу, описывающую движение маятника Капицы.
2. Постройте траектории движения маятника при разных углах начального отклонения (от 0 до $\pi/2$). Опишите результат.
3. Задайте частоту внешней силы в 10 раз больше частоты собственных колебаний.
4. Попробуйте установить амплитуду вынуждающей силы больше $\sqrt{2gl/\gamma^2}$.
5. Что происходит, если маятник запускать из положения, близкого к 180 градусам?
6. Сделайте выводы.

Движение взаимодействующих частиц. Две частицы притягиваются с силой, пропорциональной $1/r^2$. Построим траектории их движения. Физическая запись задачи содержит два векторных ДУ второго порядка:

$$m_1\frac{d^2\vec{r_1}}{dt^2} = \vec{F}_1,$$
$$m_2\frac{d^2\vec{r_2}}{dt^2} = \vec{F}_2.$$

3.2.3. Для формализации задачи и записи ее в форме (3.6) необходимо определить вектор формальных переменных:

$$\vec{x} = [t, x_1, y_1, v_{1x}, v_{1y}, x_2, y_2, v_{2x}, v_{2y}].$$

Для удобства время считается одной из переменных.

Производные соответствующих переменных дают вектор правых частей уравнения:

$$\vec{f} = \frac{d\vec{x}}{dt} = [1, v_{1x}, v_{1y}, \frac{F_{1x}}{m_1}, \frac{F_{1y}}{m_1}, v_{2x}, v_{2y}, \frac{F_{2x}}{m_2}, \frac{F_{2y}}{m_2}].$$

Задаем функцию, вычисляющую вектор производных (правая часть ДУ):

```
def force(args):
  # вектор данных t, x1, y1,v1x,v1y, x2, y2,v2x,v2y
  # производные   1,v1x,v1y,f1x,f1y,v2x,v2y,f2x,f2y

  t, x1, y1, v1x, v1y, x2, y2, v2x, v2y = args
  r = ((x1 - x2)**2 + (y1 - y2)**2)**0.5
  f1x = -(x1 - x2) / r**2
  f1y = -(y1 - y2) / r**2
  f2x = -f1x
  f2y = -f1y
  f = np.array([1., v1x, v1y, f1x, f1y, v2x, v2y, f2x, f2y])
  return f
```

Решение системы проводится стандартным методом Рунге-Кутта четвертого порядка, описанным в разделе «Решение задачи Коши» (3.2). Для этого создаем функцию шага по времени, которая возвращает значение в следующей точке. Обратите внимание, что время входит в вектор переменных, поэтому запись функции более компактна.

```
def step(f, x, h):
  kt1 = f(x) * h
  kt2 = f(x + kt1/2.) * h
  kt3 = f(x + kt2/2.) * h
  kt4 = f(x + kt3) * h
  xn = x + (kt1 + 2.*kt2 + 2.*kt3 + kt4) / 6.
  return xn
```

Задаем начальные условия, делаем много шагов и рисуем положения частиц.

```
#  вектор данных t, x1, y1, v1x, v1y, x2, y2, v2x, v2y
x = np.array([0, 1., 0, 0, 1., -1., 0, 0, -1.])
h = 0.1

py.ion()
py.xlabel('X')
py.ylabel('Y')

for nt in range(1000):
  x = step(force, x, h)
  py.plot(x[1], x[2], 'go')
  py.plot(x[5], x[6], 'bo')
```

```
13  py.draw()
```

Задания:

1. Соберите программу, описывающую движение пары частиц.
2. Постройте траектории движения частиц при разных прицельных расстояниях и разных начальных скоростях.
3. Найдите условия, при которых столкновение частиц происходит за время соударения, значительно превышающее среднее. Используйте для анимации метод, предложенный в разделе 2.8.
4. Сделайте выводы.

3.2.4. **Циклотронный резонанс.** В начале координат покоится электрон. По оси Z направлено однородное магнитное поле $B = 0.1$ Т. В момент $t = 0$ включается электрическое поле $E = Asin(2\pi ft)$, направленное по оси X; $A = 1000$ В/м, частота f — вводимый параметр.

Задания:

1. Напишите программу, описывающую движение электрона в циклотроне.
2. Нарисуйте траекторию движения $r(t)$ в плоскости XOY и график кинетической энергии $W(t)$ электрона за время $t = 3,57$ нс.
3. Постройте график кинетической энергии W электрона, которую он набирает за время $t = 3,57$ нс, в зависимости от частоты f электрического поля. Частоту f удобно выражать в ГГц, энергию W — в эВ. Интересующий диапазон частот $f = [2,5,3,1]$ ГГц. Какую максимальную энергию W_{max} набирает электрон? При какой частоте f_{max} это происходит?
4. Какую максимальную энергию W_{max} электрона (за произвольное время) можно получить при $f = 2,52$ ГГц?
5. Используя график траектории электрона, качественно объясните получаемый результат.
6. Сделайте выводы.

3.3. Задачи на движение сплошных сред и нахождение полей

Дифференциальные уравнениями в частных производных (ДУЧП) с дополнительными уравнениями, выражающими граничные и начальные условия, описывают большинство физических процессов для температурных, концентрационных, электро- магнитных и других полей. Через эти уравнения находится движение в континуальных моделях среды. В них входят производные не только по времени, но и по координатам, что от-

ражает воздействие на локальные параметры соседних точек среды. В простейшем случае одномерной задачи линейное ДУЧП второго порядка с постоянными коэффициентами имеет вид

$$a\frac{\partial^2 u(x,t)}{\partial t^2} + b\frac{\partial^2 u(x,t)}{\partial x \partial t} + c\frac{\partial^2 u(x,t)}{\partial x^2} + d\frac{\partial u(x,t)}{\partial x} + e\frac{\partial u(x,t)}{\partial t} + fu(x,t) = g, \tag{3.13}$$

где a, b, c, d, e, f, g — коэффициенты. Производные по пространственным координатам входят в уравнения симметрично, поэтому их компактно записывают через оператор набла:

$$\vec{\nabla} = \vec{i}\frac{\partial}{\partial x} + \vec{j}\frac{\partial}{\partial y} + \vec{k}\frac{\partial}{\partial z}. \tag{3.14}$$

Действие оператора на скалярную функцию даёт ее градиент, а на векторную – дивергенцию (при скалярном умножении) или ротор (при векторном умножении). Оператор Лапласа равен квадрату оператора набла.

Классификация дифференциальных уравнений в частных производных

Классификация ДУЧП проводится в соответствии с характеристическими кривыми второго порядка, которые появляются при решении данных уравнений. По соотношению коэффициентов a, b и c, входящих в уравнение 3.13, его относят к эллиптическим, параболическим или гиперболическим в данной точке. Тип ДУ определяется знаком выражения, называемого дискриминантом: $D(x,t) = b^2 - 4ac$.

- Если $D(x,t) > 0$, дифференциальное уравнение является **гиперболическим** в точке (x, t).
- Если $D(x,t) = 0$, дифференциальное уравнение является **параболическим** в точке (x, t).
- Если $D(x,t) < 0$, дифференциальное уравнение является **эллиптическим** в точке (x, t).

Если коэффициенты a, b, c постоянны и значение D не зависит от точки, то в зависимости от знака D уравнение является полностью эллиптическим, гиперболическим или параболическим. В случае, если коэффициенты не являются постоянными, для одного и того же уравнения возможно существование областей, в которых оно является уравнением разного типа.

Гиперболические уравнения Гиперболические уравнения часто называют волновыми уравнениями, т.к. с их помощью описывается распространение волн различной природы (например, упругих, электромагнитных, сдвиговых).

Общий вид гиперболического уравнения:

$$\frac{1}{v^2}\frac{\partial^2 u(x,y,z,t)}{\partial t^2} - \nabla^2 u(x,y,x,t) = f(x,y,z), \tag{3.15}$$

где v — скорость волны. К этому же типу уравнений относится нестационарное уравнение Шредингера из квантовой механики.

Параболические уравнения Параболические уравнения появляются в нестационарных задачах теплопроводности, диффузии. Иногда параболические задачи получаются из гиперболических уравнений (параболическое приближение в оптике). Уравнение теплопроводности, например, имеет вид:

$$B(x,y,z)\frac{\partial u(x,y,z,t)}{\partial t} - \nabla(A(x,y,z)\cdot\nabla u(x,y,z,t)) = f(x,y,z), \tag{3.16}$$

где u имеет смысл температуры, B — произведение плотности на удельную теплоёмкость, A — коэффициент теплопроводности, функция f в правой части — плотность источников тепла.

Эллиптические уравнения К эллиптическим уравнениям сводятся многие стационарные (установившиеся) решения параболических и гиперболических задач. Такими уравнениями описываются стационарное распределение температуры в процессе теплопереноса и стационарное распределение концентрации при диффузии. К ним приводят и другие задачи, например, о распределении электростатического поля в непроводящей среде. Простейший вид эллиптического уравнения:

$$\nabla(A(x,y,z)\cdot\nabla u(x,y,z)) = f(x,y,z), \tag{3.17}$$

где $u(x,y,z)$ — искомая функция, $A(x,y,z)$, $f(x,y,z)$ — некоторые функции независимых переменных. Функция A описывает «коэффициент распространения» величины u и может являться тензорной величиной в случае анизотропной среды. Функция f (функция источников) — скалярная величина, показывающая плотность «скорости появления» величины u в единице объёма. В качестве величин, входящих в это уравнение, в зависимости от смысла могут использоваться температура, коэффициент теплопроводности, плотность тепловых источников или потенциал электрического поля, диэлектрическая проницаемость и плотность зарядов и т.д

Начальные и граничные условия

Из курса высшей математики известно, что дифференциальные уравнения, как правило, имеют бесконечное множество решений. Это связано с появлением в процессе интегрирования констант, при любых значениях которых решение удовлетворяет исходному уравнению. Задачи физики связаны с нахождением зависимости от координат и времени определённых физических величин, которые, безусловно, должны удовлетворять требованиям однозначности, конечности и непрерывности. Иными словами, любая задача физики предполагает поиск единственного решения (если оно вообще существует). Поэтому математическая формулировка физической задачи должна помимо основных дифференциальных уравнений, описывающих искомые функции, включать дополнительные уравнения (дифференциальные или алгебраические), описывающие искомые функции на границах рассматриваемой области в любой момент времени и во всех внутренних точках области в начальный момент времени. Эти дополнительные уравнения называют соответственно граничными и начальными условиями задачи. Условия, относящиеся к точкам пространства, называются граничными. Обычно это неизменные условия, накладываемые на значение функции или на ее производную (поток через границу) на границе рассматриваемой области. Начальные условия – значения физической величины в начальный момент времени. Только после задания обоих типов условий можно получить описание развития процесса во времени. Для ДУЧП редко решают задачи, когда условия внутри области заданы для различных моментов времени, т.к. это сильно усложняет и без того непростую процедуру поиска решения.

Методы дискретизации ДУЧП

К сожалению, аналитическое решение уравнений математической физики возможно лишь для весьма ограниченного круга задач. В большинстве случаев решение дифференциальных уравнений в частных производных возможно только с использованием численных методов. Суть данных методов состоит в дискретизации дифференциальных уравнений, то есть представлении производных в виде приближенных выражений, что позволяет преобразовать дифференциальные уравнения в системы алгебраических уравнений. Для этого рассматриваемая область покрывается сеткой, а все переменные заменяются сеточными функциями. Иными словами, значения функции находятся не для всего множества точек области, а лишь для некоторого конечного подмножества. Число алгебраических уравнений в полученной системе (размерность дискретной задачи) определяется

произведением числа точек координатной сетки на количество независимых переменных в исходных дифференциальных уравнениях. При решении нестационарных задач помимо координатной сетки вводится сетка времени. Точки, относящиеся к одному моменту времени, называют *слой*. Нахождение значения функции на следующем слое похоже на решение задачи Коши для обыкновенных дифференциальных уравнений.

Выбор метода решения полученной системы алгебраических уравнений определяется ее размерностью и характером (линейный или нелинейный). Для решения систем линейных алгебраических уравнений широко используют метод исключения Гаусса, метод LU-разложения и др. Для решения систем нелинейных алгебраических уравнений и линейных систем больших размерностей используют итерационные методы Якоби, Зейделя, Ньютона-Рафсона и др.

Метод конечных разностей Метод конечных разностей предполагает дискретизацию функций на некоторой сетке. Обычно используют сетки совпадающие с системой координат, т.к. при таком выборе частные производные имеют простой вид. Для декартовой системы координат ячейки сетки представляют собой прямоугольники (или параллелепипеды), а выражения для производных записывают через правую или левую разность:

$$\frac{\partial u(x,t)}{\partial x} = \frac{u(x_0+h,t_0)-u(x_0,t_0)}{h}, \tag{3.18}$$

$$\frac{\partial u(x,t)}{\partial x} = \frac{u(x_0,t_0)-u(x_0-h,t_0)}{h}. \tag{3.19}$$

Лучше использовать центральные разности, которые для первой и второй производных имеют вид:

$$\frac{\partial u(x,t)}{\partial x} = \frac{u(x_0+h,t_0)-u(x_0-h,t_0)}{2h}, \tag{3.20}$$

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{u(x_0+h,t_0)-2u(x_0,t_0)+u(x_0-h,t_0)}{h^2}. \tag{3.21}$$

Аналогично строятся смешанные производные и производные высших порядков.

Для эллиптических уравнений выбирается сетка по пространственным координатам, а если уравнение параболическое или гиперболическое, то и по времени. ДУЧП после такой замены превращается в систему алгебраических уравнений, решение которой проводится стандартными методами. Размер системы уравнений определяется плотностью сетки, а увеличение точности достигается уменьшением шага разбиения.

Для однородного эллиптического уравнения в качестве переменных выступают пространственные координаты:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0.$$

Разностное уравнение имеет вид:

$$\lambda^2 u_{i+1,j} + \lambda^2 u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 2(1+\lambda^2) u_{i,j} = 0,$$

где отношение шагов разбиения по координатам y и x обозначено $\lambda = k/h$. Для удобства представления разностные схемы уравнений записывают на сетке, в узлах которой отмечают весовые коэффициенты, которые нужно взять, чтобы линейная комбинация значений давала ноль. Для эллиптического уравнения схема представлена на рисунке 3.3. Решение полученной

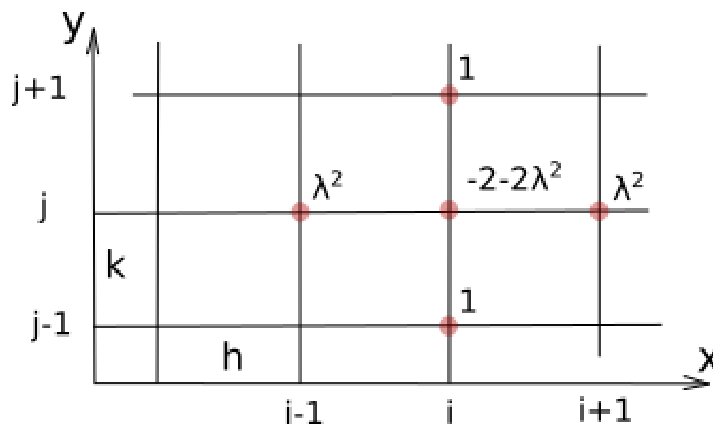


Рис. 3.3.: Графическое представление разностной схемы для эллиптического уравнения.

системы уравнений может быть проведено методом последовательных приближений. Для этого задаётся начальное приближение во всех узлах сетки, а затем значения в каждом узле пересчитываются через значение в соседних узлах по схеме, приведённой выше. Для такого метода выполнено условие сходимости, поэтому в результате получаем искомое решение.

Для решения гиперболического уравнения

$$\Delta u(\vec{r},t) - \frac{\partial^2 u(\vec{r},t)}{\partial t^2} = 0$$

необходимо задать граничные условия для всех моментов времени: $u(t)|_\Gamma$ и начальные условия для значения функции и ее производной: $u(\vec{r},0)$, $\left.\frac{\partial u(\vec{r},t)}{\partial t}\right|_{t=0}$. Одномерное разностное гиперболическое уравнение можно записать в виде:

$$-\lambda^2 u_{i+1,j} - \lambda^2 u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + 2(\lambda^2 - 1) u_{i,j} = 0,$$

где вторая переменная имеет смысл времени, а отношение шага по времени к шагу по координате обозначено $\lambda = k/h$. Графическая схема полученного метода приведена на рисунке 3.4. Начальные условия при $t = 0$

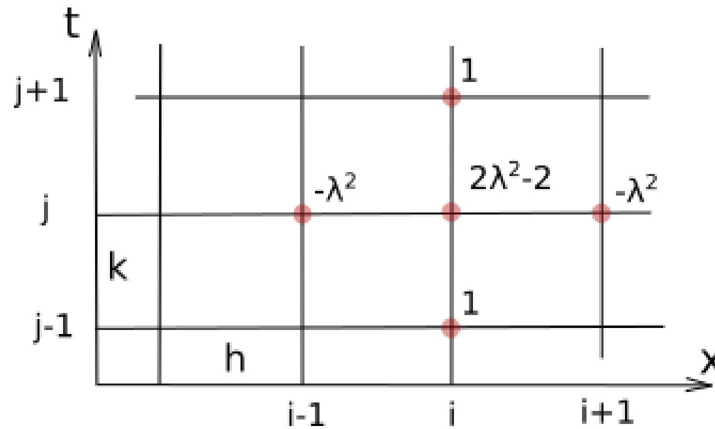


Рис. 3.4.: Графическое представление разностной схемы для гиперболического уравнения

задают значения функции на нижнем слое точек ($j = 1$), а ее производные позволяют вычислить значения функции на втором слое точек ($j = 2$). Граничные условия обеспечивают полноту информации для каждого слоя. Решение данной разностной схемы может быть выполнено в явном виде. Достаточно выразить значение функции на следующем слое через известные значения на предыдущих:

$$u_{i,j+1} = \lambda^2 u_{i+1,j} + \lambda^2 u_{i-1,j} - u_{i,j-1} + 2(1 - \lambda^2) u_{i,j},$$

где $\lambda = k/h$.

Для данной схемы условием устойчивости является $k < h$, т.е. шаг по координате должен быть больше, чем шаг по времени. Это не совсем удобно, т.к. не позволяет вести интегрирование с большим шагом.

Для параболического уравнения

$$\Delta u(\vec{r}, t) - \frac{\partial u(\vec{r}, t)}{\partial t} = 0$$

так же, как и для гиперболических, задаются начальные и граничные условия. Из-за первого порядка таких уравнений по времени в качестве начальных условий достаточно задать только значения функции $u(\vec{r}, 0)$.

Простейшая разностная схема для одномерного параболического уравнения:

$$u_{i,j+1} - \lambda u_{i+1,j} - \lambda u_{i-1,j} - (1 - 2\lambda) u_{i,j} = 0,$$

где $\lambda = k/h$. Её графическая схема показана на рисунке 3.5 а). Это *явная* схема, т.к. значения функции на $j+1$-м слое сразу находится из значений на j слое. Явная схема устойчива при $k < h^2/2$, т.е. шаг по времени должен быть меньше, чем квадрат сетки по координате. Это очень сильное условие, которое делает явную схему не работоспособной. Неустойчивость решения вызвана слабой связью между соседними точками на следующем слое. При потере устойчивости возникает знакопеременное решение с близкими по модулю значениями.

Для параболического уравнения обычно используются *неявные* схемы вычислений. Простая неявная схема, которая устойчива при всех параметрах:

$$-u_{i,j} - \lambda u_{i+1,j+1} - \lambda u_{i-1,j+1} + (1+2\lambda)u_{i,j+1} = 0.$$

Ее схема приведена на рисунке 3.5 b). Ещё одна неявная схема носит название «метод Кранка–Николса», который тоже устойчив при всех параметрах. В нем использована лучшая аппроксимация производных:

$$\frac{\lambda}{2}(u_{i-1,j} + u_{i+1,j} + u_{i-1,j+1} + u_{i+1,j+1}) - (\lambda+1)u_{i,j+1} - (\lambda-1)u_{i,j} = 0.$$

Графическое представление схемы метода Кранка–Николса приведено на рисунке 3.6.

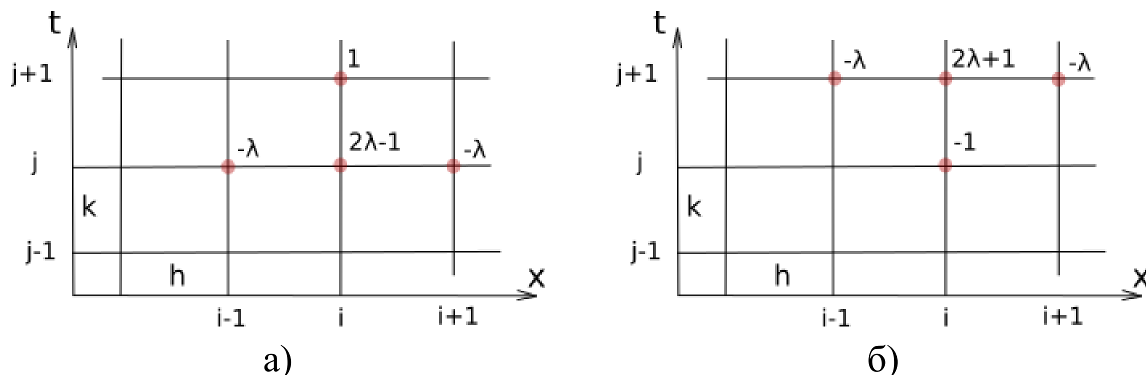


Рис. 3.5.: Графическое представление разностных схем для параболического уравнения. а) – явная схема, b) – неявная схема.

Неявные схемы вычислений приводят к необходимости решать на каждом шаге систему линейных уравнений типа

$$au_{i-1,j+1} + bu_{i,j+1} + cu_{i+1,j+1} = d,$$

но большой плюс состоит в том, что эта системы трехдиагональна и хорошо решается методом прогонки, трудоёмкость которого линейно растёт с увеличением размера.

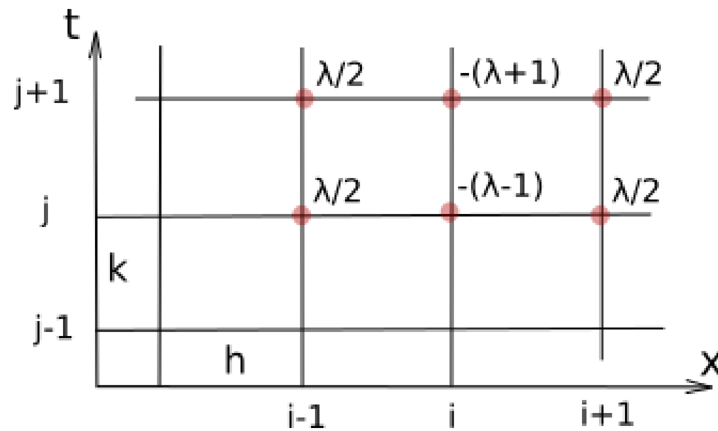


Рис. 3.6.: Графическое представление метода Кранка–Николса для параболического уравнения.

Примеры и задания

3.3.1. Задача о «Квантовых часах»[1]

Построить численное решение временного уравнения Шредингера (одномерный случай):

$$-\frac{\hbar^2}{2m}\Delta\psi(\vec{r},t)+E(\vec{r})\psi(\vec{r},t)=i\hbar\frac{\partial}{\partial t}\psi(\vec{r},t), \qquad (3.22)$$

для симметричного потенциала в виде двух одинаковых потенциальных ям. Простейший вариант — две прямоугольные ямы. Учтите, что ямки должны быть достаточно близко друг к другу (ширина барьера существенно меньше ширины ямок), чтобы вероятность туннелирования через барьер между ними не была катастрофически мала. Первоначально частица находится в одной из ям. При таких условиях волновая функция будет со временем «перетекать» из одной ямки в другую и обратно через одинаковые промежутки времени, подобно маятнику часов.

Решение Для решения необходимо задать основные параметры дискретизации

```
dx = 0.2
xmax = 40.0
sp_x = np.arange(-xmax,xmax,dx)

dt = 0.02
sp_t = np.arange(0.0,100.0,dt)
```

Задаем симметричный потенциал в виде двух одинаковых потенциальных ям:

[1] Задача предлагается на кафедре 32 МИФИ.

```python
b = 1.0
a = 1.0
v0 = -1.0
def v(x):
   return (v0/np.cosh((x-(b+a/2.0))/a)**2 +
           v0/np.cosh((x+(b+a/2.0))/a)**2.)
V = v(sp_x).astype('complex')
```

Начальные условия – частица находится в одной из ямок. Можно задать волновую функцию, например распределением Гаусса, в пределах одной ямки:

```python
#Покоящаяся частица в правой яме
init = np.exp(-(sp_x-(b+a/2))**2 / (2*a)**2).astype('complex')
```

Самое чёткое проявления эффекта будет, если взять начальное условие как сумму основного и первого возбуждённого состояний для выбранного потенциала. Разумеется, перед суммированием волновые функции необходимо отнормировать:

```python
#Нормировка
init = init/np.dot(init, np.conj(init))**0.5
```

Можно выбрать любую разностную схему, однако явная схема неустойчива при решении временного уравнения Шредингера. Задачу можно попытаться выполнить по неявной схеме, но в этом случае Вы можете столкнуться с проблемой, что нормировка будет со временем уменьшаться. Это можно исправить, проводя перенормировку волновой функции после каждого шага по времени, но в некотором смысле это «жульничество». Для решения этой задачи лучше всего воспользоваться схемой Дюфорта–Франкеля, или схемой ромба. По своей сути эта схема напоминает явную, но вместо двух слоёв по времени в этой схеме задействовано три. В отличие от простейшей явной разностной схемы, здесь на каждом шаге вместо средней точки по координате нужно взять полусумму следующего и предыдущего шагов по времени в этих точках:

$$\Delta\psi \to [\psi(x+1,t)+\psi(x-1,t)-\psi(x,t-1)-\psi(x,t+1)]/h^2, \quad (3.23)$$

$$\frac{d\psi}{dt} \to [\psi(x,t+1)-\psi(x,t-1)]/(2\tau), \quad (3.24)$$

$$\psi \to [\psi(x,t+1)+\psi(x,t-1)]/2, \quad (3.25)$$

где h — шаг по координате, τ — шаг по времени. Такая схема всегда остается явной относительно $\Psi(x,t+1)$ и является устойчивой при любых

h и L.[1]. Реализация вычислений для одного слоя приведена в (крайне ужасном) листинге.

```
for x in range(size(sp_x)):
    # точка в середине интервала
    if (x!=0) and (x!=size(sp_x)-1):
        temp = (2j*dt/(dx**2))*(u[t-1,x+1] + \
                u[t-1, x-1] - u[t-2, x])
        temp += -2j*dt*V[x]*u[t-1,x] + u[t-2,x]
        u[t,x] = temp/(1. + 2j*dt/dx**2)
    # крайняя точка интервала
    if (x==0):
        temp = (2j*dt/dx**2)*(u[t-1,x+1] - u[t-2, x])
        temp += -2j*dt*V[x]*u[t-1,x] + u[t-2,x]
        u[t,x] = temp/(1. + 2j*dt/dx**2)
    # крайняя точка интервала
    if (x==size(sp_x) - 1):
        temp = (2j*dt/dx**2)*(u[t-1, x-1] - u[t-2, x])
        temp += -2j*dt*V[x]*u[t-1,x] + u[t-2,x]
        u[t,x] = temp/(1. + 2j*dt/dx**2)
```

Как видно, одного начального условия недостаточно, чтобы начать пользоваться этой схемой, так как нам нужно минимум 2 слоя по времени. Второй слой можно взять из первого по обычной явной схеме. Несмотря на то, что эта схема неустойчива и со временем решение «разваливается», один шаг по ней можно сделать без заметных последствий.

```
u = np.zeros((size(sp_t),size(sp_x)), dtype=dtype('complex'))
u[0,:] = init

#first step - explicit scheme
for x in range(size(sp_x)):
  # точка в середине интервала
  if (x!=0) and (x != size(sp_x)-1):
    u[1,x] = u[0, x]+(1j)*dt*(u[0,x+1] - \
             2*u[0,x]+u[0,x-1])/dx**2 - 1j*dt*V[x]*u[0,x]
  # крайняя точка интервала
  if x==0:
    u[1,x] = u[0,x]+(1j)*dt*(u[0,x+1] - 2*u[0,x])/dx**2 \
             - 1j*dt*V[x]*u[0,x]
  # крайняя точка интервала
  if x==size(nx)-1:
    u[1,x] = u[0,x]+dt*(1j)*(-2*u[0,x] + u[0,x-1])/dx**2 \
             - 1j*dt*V[x]*u[0,x]
```

Анимация решения:

[1]Строгое доказательство факта можно найти в книге А.А. Самарского: Самарский А. А. Введение в теорию разностных схем. - М.: Наука, 1971. - 552 с.

```
from matplotlib.animation import FuncAnimation

mult = 15        # рисуем функцию через mult шагов по времени

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(sp_x,V)  # потенциал
line, = ax.plot(sp_x, np.abs(init)**2)

def animate(i):  # изменяем нарисованную линию
    line.set_ydata(np.abs(u[mult*i,:])**2)
    return line,

ani = FuncAnimation(fig, animate, interval=500)
plt.show()
```

Задания:

- Получите численную схему для решения уравнения из приведённых выше уравнений, нарисуйте ее графическое представление.
- Соберите программу для решения задачи о квантовых часах.
- Проведите исследование и постройте зависимость периода часов от ширины ям и расстояния между ними.
- На примере случая, когда начальные условия являются суперпозицией основного и первого возбуждённого состояний (считается что волновые функции и энергии известны) обоснуйте теоретически, откуда возникает эффект квантовых часов.

3.3.2. **Решение уравнения Кортевега–де Фриза.**

Уравнение Кортевега–де Фриза — нелинейное уравнение в частных производных третьего порядка, играющее важную роль в теории нелинейных волн, в основном гидродинамического происхождения. Уравнение имеет вид:

$$\frac{\partial u(x,t)}{\partial t} + 6u(x,t)\frac{\partial u(x,t)}{\partial x} + \frac{\partial^3 u(x,t)}{\partial x^3} = 0 \tag{3.26}$$

1. Предложите разностную схему для решения данного уравнения.
2. Возьмите в качестве начальных данных уравнение уединённого солитона $u(x,0) = \frac{2k^2}{\cosh^2[k(x-x_0)]}$.
3. Найти $u(x,t)$ при $t = 150$, покажите, что форма решения сохраняется.
4. Постройте картину движения солитона.
5. Возьмите в качестве начальных данных функцию Гаусса $u(x,0) = a\exp(-\frac{x^2}{b^2})$.
6. Постройте картину движения солитона до $t = 150$.

3.3.3. **Нелинейное волновое уравнение.**

Решить задачу Коши для уравнения

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{\partial^2 u(x,t)}{\partial x^2} - \sin(u(x,t)) \quad (3.27)$$

на решётке от 0 до 10 с шагом $dx = 0.005$.

1. Возьмите в качестве начальных данных $u(x, t_0)$ и $u'(x, t_0)$ для t_0 волновой пакет, перемещающийся со скоростью v.
2. Найти $u(x, t)$ при $t = 150$.
3. Подберите скорость так, чтобы пакет перемещался с одного края решётки до другого, но не выходил за ее пределы.
4. Постройте картину движения пакета.

3.3.4. **Квантовая частица.**

Решить задачу Коши для уравнения Шредингера

$$i\frac{\partial \psi(x,t)}{\partial t} = -\frac{\partial^2 \psi(x,t)}{\partial x^2} + V(x)\psi(x,t)$$

с потенциалом: $V(x) = u_0\delta(x+a) + u_0\delta(x-a)$.

1. Начальные условия выбрать в виде Гауссового пакета с центром в x_0, шириной a и импульсом p_0: $u(x,t) = \exp(-\frac{(x-x_0)^2}{2a^2} + i \cdot p_0 x)$.
2. Подобрать диапазон изменения координаты и диапазон изменения времени t так, чтобы можно было проследить всю картину рассеяния пакета.
3. Подобрать ширину пакета и величину потенциала так, чтобы были видны колебания метастабильного уровня и экспоненциальные хвосты пакетов после рассеяния.
4. Постройте картину движения пакета.

3.3.5. **Нелинейное уравнение Шредингера.**

Для нелинейного уравнения Шредингера:

$$iu_t + u_{xx} + \nu|u|^2 u = 0, \quad (3.28)$$

при значении параметра $\nu > 0$ допустимы уединённые волны в виде:

$u(x,t) = \left(\sqrt{\frac{2\alpha}{\nu}}\right)\cosh^{-1}\left(\sqrt{\alpha}(x-Ut)\right)e^{i(rx-st)}$, где r, s, α, U — некоторые постоянные, связанные соотношениями: $U = 2r$, $s = r^2 - \alpha$.

1. Напишите программу для решения указанного уравнения.
2. Проверьте, что волны в указанном виде сохраняются при движении.

Заключение

Рассмотренные в пособии вопросы являются основой для грамотного использования компьютера специалистами в области вычислений. Однако это лишь начальный шаг при освоении возможностей, открывающихся для ученых при работе с ЭВМ. Дальнейшее движение, на наш взгляд, имеет три направления:

1. Если вам нравится программирование, то можно обратить внимание на то, что Python позволяет просто решать задачи, связанные с созданием графических интерфейсов пользователя, разработкой сетевых приложений; он интегрирован во многие прикладные программные пакеты как средство для их гибкой настройки. Поэтому, совершенствуя навыки программирования, можно перейти к решению задач, напрямую не связанных с физикой.
2. Если вы планируете стать разработчиком научного программного обеспечения, стоит уделить время знакомству с разделами, не затронутыми в данном учебном пособии. В первую очередь это написание документации к программам и знакомство с системами автоматического документирования, например, Sphinx-doc. Вам также не обойтись без изучения систем совместной разработки программного обеспечения и контроля версий, таких, как SVN или Git.
3. Если ваша цель — использование компьютера для личных целей в качестве «большого калькулятора», то для углубления знаний в области численного анализа различного рода физических задач можно порекомендовать книгу Гулда и Тобочника[1] с доступным изложением идей практически всех разделов вычислительной физики. Математическая сторона вычислений представлена в серии книг Самарского А.А.[2], которые стали классическими учебниками–справочниками.

В пособии в основном использованы наиболее простые задачи, возникающие в общем курсе физики. При попытке численного исследования более сложных систем можно столкнуться с целым рядом не решенных или малоизученных вопросов, исследование которых может стать началом серьезной научной работы.

[1] Гулд Х., Тобочник Я. Компьютерное моделирование в физике: Перевод с англ. - М.: Мир, 1990.

[2] Самарский А. А. Введение в теорию разностных схем. - М.: Наука, 1971. - 552 с.; Самарский А. А. Введение в численные методы. 1987; Самарский А. А., Михайлов А. П. Математическое моделирование: Идеи. Методы. Примеры. - М.: Физматлит, 2005 (5-е изд.).

Замечания и предложения по пособию направляйте на адреса авторов: Соболев Андрей Николаевич – andrey@physics.susu.ac.ru; Воронцов Александр Геннадьевич – sas@physics.susu.ac.ru. Все полученные отзывы будут учтены при подготовке второго издания учебного пособия.

A. Справочник команд Python

A.1. Математические операции

Операция	Описание
`a + b`	сумма
`a - b`	разность
`a * b`	умножение
`a / b`	деление (если a и b — целые, то целочисленное)[1]
`a // b`	целочисленное деление[2]
`a ** b`	возведение в степень
`abs(a)`	модуль числа
`-a`	изменение знака числа

[1] В версиях Python начиная с 3.0 прямой слеш (/) обозначает дробное деление безотносительно типов операндов. Такого поведения можно добиться и от Python 2.7, вставив в преамбулу файла строку `from __future__ import division`.

[2] Если оба операнда — вещественные числа, в результате операции получится вещественное число с дробной частью, равной нулю.

A.2. Логические операции

Операция	Описание
`a or b`	Логическое «или» (a или b)
`a and b`	Логическое «и» (a и b)
`not a`	Отрицание (не a)
`a in b`	Принадлежность (a содержится в b)
`a not in b`	То же, что `not a in b`
`a is b`	Идентичность (a и b — один и тот же объект)
`a is not b`	То же, что `not a is b`

A.3. Логические условия

Условие	Описание
`a == b`	равенство
`a != b` или `a<>b`	неравенство
`a < b`	Условие «a меньше b»
`a > b`	Условие «a больше b»
`a <= b`	Условие «a меньше либо равно b»
`a >= b`	Условие «a больше либо равно b»
`a > b > c`	То же, что `(a > b) and (b > c)`

A.4. Доступ к элементам последовательностей

В примерах следующей таблицы `a = [1,2,3,4,5]`.

Команда	Описание	Пример
`a[i]`	Доступ к i-му элементу последовательности (начиная с нулевого)	`>>> a[2]` `3`
`a[-i]`	Доступ к i-му элементу последовательности с конца	`>>> a[-2]` `4`
`a[i:j]`[1]	Элементы последовательности, начиная с i-го и заканчивая $(j-1)$-ым (элемент j не включается в срез)	`>>> a[2:4]` `[3, 4]`
`a[i:j:k]`[1]	То же, что выше, с шагом k (каждый элемент, индекс которого в срезе кратен k)	`>>> a[1:4:2]` `[2, 4]`

[1] В *срезах* начальный или конечный индекс среза может быть опущен; в таком случае срез берется с самого начала (или до самого конца) последовательности. Например, `a[:3:2]` выдаст каждый четный элемент с начала последовательности до второго элемента включительно.

A.5. Функции для работы с последовательностями

В примерах следующей таблицы `a = [1,2,3,4,5]`, `b = [10,2,8]`.

Команда	Описание	Пример
`len(a)`	Длина последовательности	`>>> len(a)` `5`
`a + b`	Конкатенация последовательностей	`>>> a + b` `[1, 2, 3, 4, 5, 10, 2, 8]`
`n * a`[1]	Последовательность a, повторенная n раз	`>>> 2 * b` `[10, 2, 8, 10, 2, 8]`

`min(a), max(a)`	Минимальный (или максимальный) элемент последовательности	`>>> min(a)`
		`1`
		`>>> max(b)`
		`10`

[1] При $n < 0$ возвращается пустая последовательность.

A.6. Форматирование строк

Строка-заполнитель содержит два или более символа и состоит из следующих компонентов, идущих по порядку:

1. Символ % — начало строки-заполнителя;
2. Минимальная ширина поля — минимальное количество символов, требуемых для вывода переменной (необязательно)
3. Точность — пишется через '.', после которой стоит число, описывающее требуемое количество знаков после запятой (необязательно);
4. Модификатор длины (необязательно) — число;
5. Тип переменной (см. в таблице).

Формат	**Соответствующий тип**
`d`	Целое число (int)
`o`	Целое число в восьмеричном формате
`x`	Целое число в шестнадцатиричном формате
`f`	Дробное число (float)
`e`	Дробное число в инженерном формате
`c`	Один символ
`s`	Строка (string)

B. Справочник команд Numpy

B.1. Создание массивов из имеющихся данных

- a – объект или массив
- object – любой объект с упорядоченными данными
- dtype – тип данных (если не указан определяется по данным объекта)
- copy – да/нет, создать копию данных
- order – {'C', 'F', 'A'}– порядок размещения элементов в памяти (Си, Фортран, любой)
- ndmin – минимальное число измерений (добавляет пустые массивы по недостающим измерениям)
- buffer – объект буфера
- count – число данных для чтения
- offset – отступ от начала
- file – объект файла
- sep – шаг чтения файла
- string – строка
- function – функция. Вызывается `function(i,j,k,**kwargs)`, где i,j,k – индексы ячейки массива
- shape – форма массива
- **kwargs – словарь параметров для функции
- fname – имя файла
- comments – символ коментария
- delimiter – разделитель данных

Команда	**Описание**
array(object[, dtype, copy, order, subok, ndmin])	Создать массив из последовательности
asarray(a[, dtype, order])	Преобразовать в массив
ascontiguousarray(a[, dtype])	Размещает в памяти непрерывный массив(порядок данных как в Си)
asmatrix(data[, dtype])	Представить данные как матрицу
copy(a)	Возвращает копию объекта
frombuffer(buffer[, dtype, count, offset])	Использует буфер, как одномерный массив
fromfile(file[, dtype, count, sep])	Создает массив из данных файла
fromfunction(function, shape, **kwargs)	Создает и заполняет массив значениями функции от индексов элемента
fromiter(iterable, dtype[, count])	Создает одномерный массив из итератора
fromstring(string[, dtype, count, sep])	Создает одномерный массив из строки
loadtxt(fname[, dtype, comments, delimiter, ...])	Создает массив из данных текстового файла
Команда	**Описание**

B.2. Создание массивов определённого вида

- shape – форма массива
- dtype – тип данных
- order – порядок размещения данных(Си, Фортран)
- a – объект типа массива
- N – число строк
- M – число столбцов
- k – задает диагональ (к=0 – главная, к $>$ 0 – смещение вверх, к $<$ 0 – смещение вниз)
- x – одномерный массив или список

Команда	Описание	Пример

`empty(shape [, dtype, order]); empty_like(a [, dtype, order, subok])`	выделяет место без инициализации (случайные числа)	`>>> np.empty([2, 2], dtype=int)`
		`array([[-10737418, -10679491],`
		`       [ 4960419, 192497]])`

`eye(N [, M, k, dtype])`	двухмерный диагональный со сдвигом	`>>> np.eye(3, k=1)`
		`array([[ 0., 1., 0.],`
		`       [ 0., 0., 1.],`
		`       [ 0., 0., 0.]])`

`identity(N [, dtype])`	единичная матрица (квадратная)	`>>> np.identity(3)`
		`array([[ 1., 0., 0.],`
		`       [ 0., 1., 0.],`
		`       [ 0., 0., 1.]])`

`ones(shape [, dtype, order]); ones_like(a [, out])`	все единицы	
`zeros(shape [, dtype, order]); zeros_like(a [, dtype, order, subok])`	все нули	

`tri(N [, M, k, dtype])`	нижняя треугольная (из единиц)	`>>> np.tri(3, 5, 2, dtype=int)`
		`array([[1, 1, 1, 0, 0],`
		`       [1, 1, 1, 1, 0],`
		`       [1, 1, 1, 1, 1]])`

`tril(a [, k])`	вырезание нижней треугольной	`>>> np.tril([[1,2,3], [4,5,6], [7,8,9], [10,11,12]], -1)`
		`array([[ 0, 0, 0],`
		`       [ 4, 0, 0],`
		`       [ 7, 8, 0],`
		`       [10, 11, 12]])`

Продолжение на следующей странице

Таблица B.1 – *Продолжение*

Команда	Описание	Пример

`triu(a [, k])`	вырезание верхней треугольной	`>>> np.triu([[1,2,3],[4,5,6], [7,8,9], [10,11,12]], -1)`
		`array([[ 1, 2, 3],`
		`[ 4, 5, 6],`
		`[ 0, 8, 9],`
		`[ 0, 0, 12]])`

`diag(a [, k])`	вырезает диагональ или создаёт двумерную диагональную матрицу	
`diagflat(a [, k])`	двумерная диагональная матрица со всеми элементами из а	

`vander(x [, N])`	создает определитель Ван-Дер-Монда	`>>> x = np.array([1, 2, 3, 5])`
		`>>> N = 3`
		`>>> np.vander(x, N)`
		`array([[ 1, 1, 1],`
		`[ 4, 2, 1],`
		`[ 9, 3, 1],`
		`[25, 5, 1]])`

`mat(data [, dtype])`	преобразует данные в матрицу	

B.3. Создание сеток

- start – начало
- stop – окончание (для «arrange» по умолчанию НЕ включается, для остальных функций - включается)
- step – шаг
- num – число точек в выходном наборе
- endpoint – да/нет, включать крайнюю точку в набор данных
- retstep – да/нет, добавить в данные величину интервала
- x, y – одномерные массивы разбиения для осей.
- base – основание логарифма

Команда	Описание	Пример
`arange([start,] stop, [step,] dtype=None)`	Похоже на `range()`	`>>> np.arange(3.0)` `array([ 0., 1., 2.])`

`linspace(start, stop [, num, endpoint, retstep])`	Равномерный набор точек	`>>> np.linspace(2.0, 3.0,`
		`↪ num=5)`
		`array([ 2., 2.25, 2.5,`
		`↪ 2.75, 3.])`

`logspace(start, stop [, num, endpoint, base])`	Логарифмический набор точек	`>>> np.logspace(2.0, 3.0,`
		`↪ num=4, base=2.0)`
		`array([ 4., 5.03968,`
		`↪ 6.34960, 8.])`

`meshgrid(x, y)`	два вектора, описывающих точки ортогональной сетки	`>>> X, Y = np.meshgrid(`
		`↪ [1,2,3], [4,5,7])`
		`>>> X`
		`array([[1, 2, 3],`
		`[1, 2, 3],`
		`[1, 2, 3]])`
		`>>> Y`
		`array([[4, 4, 4],`
		`[5, 5, 5],`
		`[7, 7, 7]])`

`mgrid`	полный набор данных, описывающий многомерную равномерную ортогональную сетку (X,Y) или (X,Y,Z). Аргументы по каждому измерению: (start:stop:step). Если step — мнимое (5j), то задается количество интервалов разбиения	`>>> np.mgrid[0:5:3j,`
		`↪ 0:5:3j]`
		`array([[[0., 0., 0.],`
		`[2.5, 2.5, 2.5],`
		`[5., 5., 5.]],`
		`[[0., 2.5, 5.],`
		`[0., 2.5, 5.],`
		`[0., 2.5, 5.]]])`

Продолжение на следующей странице

Таблица B.2 – *Продолжение*

Команда	**Описание**	**Пример**

`ogrid`	сокращенный набор данных, описывающий многомерную равномерную ортогональную сетку (X,Y) или (X,Y,Z). Аргументы по каждому измерению: (start:stop:step). Если step — мнимое (5j), то задается количество интервалов разбиения	`>>> ogrid[0:5,0:5]`
		`[array([[0],`
		`        [1],`
		`        [2],`
		`        [3],`
		`        [4]]),`
		` array([[0, 1, 2, 3, 4]])]`

B.4. Трансформации массива без изменения элементов

- a – объект типа массив
- newshape – форма массива
- order – порядок размещения элементов в памяти
- axis, axis1, axis2 – индексы осей массива
- shift – количество шагов сдвига
- k – количество поворотов
- repeats – количество повторов

Команда	**Описание**
`resize(a, new_shape)`	возвращает новый массив заданной формы (если элементов не хватает, то заполняется циклически)
`reshape(a, newshape [, order])`	новая форма для данных (полный размер обязан совпадать)
`ravel(a [, order])`	возвращает новый одномерный массив
`a.flat`	итератор по массиву (вызывается как метод)
`a.flatten( [order])`	копия массива без формы (вызывается как метод)

Продолжение на следующей странице

Таблица В.3 – *Продолжение*

Команда	**Описание**
`swapaxes(a, axis1, axis2)`	меняет две оси в массиве
`a.T`	то же что и транспонирование (если размерность=1, то не изменяется)
`transpose(a [, axes])`	транспонирует массив (переставляет измерения)
`fliplr(a)`	симметрично отображает массив относительно вертикальной оси (право-лево)
`flipud(a)`	симметрично отображает массив относительно горизонтальной оси (верх-низ)
`roll(a, shift [, axis])`	циклический сдвиг элементов вдоль выбранного направления
`rot90(a [, k])`	поворот массива против часовой стрелке на 90 градусов
`tile(a, repeats)`	создаёт массив повторением заданного
`repeat(a, repeats[, axis])`	повторяет элементы массива

В.5. Слияние и разделение массивов

- tup – кортеж массивов
- axis – индекс оси
- a – массив
- indices_or_sections – размер порции при разделении

Команда	**Описание**
`column_stack(tup)`	собирает одномерные массивы - столбцы в двухмерный
`concatenate(tup [, axis])`	соединяет последовательность массивов вместе
`dstack(tup)`	собирает массивы «по глубине» (по третьей оси)
`hstack(tup)`	собирает массивы «по горизонтали» (по столбцам)

Продолжение на следующей странице

Таблица В.4 – *Продолжение*

Команда	**Описание**
`vstack(tup)`	собирает массивы «по вертикали» (по строкам)
`array_split(a, indices_or_sections [, axis])`	разделяет массив по порциям
`dsplit(a, indices_or_sections)`	разделяет массив «по глубине» (по третьей оси)
`hsplit(a, indices_or_sections)`	разделяет массив «по горизонтали» (по столбцам)
`split(a, indices_or_sections [, axis])`	разделяет массив на части равной длины
`vsplit(a, indices_or_sections)`	разделяет массив «по вертикали» (по строкам)

В.6. Алгебраические функции для массивов

- x – массив
- out, out1, out2 – массивы нужного размера и формы для получения результата. Если нет, то создаетя новый массив.

Функция	**Описание**
`isreal(x)`	проверка на действительность (по элементам)
`iscomplex(x)`	проверка на комплексность (по элементам)
`isfinite(x [, out])`	проверка элементов на числовое значение (не бесконечность и не «не число»).
`isinf(x [, out])`	проверка на бесконечность (по элементам)
`isnan(x [, out])`	проверка аргумента на «не число» (NaN), результат – логический массив
`signbit(x [, out])`	истина, если установлен бит знака (меньше нуля)

Продолжение на следующей странице

Таблица В.5 – *Продолжение*

Функция	**Описание**
`copysign(x1, x2 [, out])`	меняет знак x1 на знак x2 (по элементам)
`nextafter(x1, x2 [, out])`	следующее после x1 в направлении x2 число, представимое в виде с плавающей точкой (по элементам)
`modf(x [, out1, out2])`	дробная и целая часть числа
`frexp(x [, out1, out2])`	разделение числа на нормированную часть и степень
`absolute(x [, out])`	Calculate the absolute value element-wise.
`rint(x [, out])`	округление элементов массива
`trunc(x [, out])`	отбрасывание дробной части (по элементам)
`floor(x [, out])`	целая часть
`ceil(x [, out])`	минимальное целое большее числа
`sign(x [, out])`	знаки элементов
`conj(x [, out])`	комплексное сопряжение (по элементам)
`exp(x [, out])`	экспонента (по элементам)
`exp2(x [, out])`	2**элемент (по элементам)
`log(x [, out])`	натуральный логарифм (по элементам)
`log2(x [, out])`	двоичный логарифм (по элементам)
`log10(x [, out])`	десятичный логарифм (по элементам)
`expm1(x [, out])`	exp(x) - 1 (по элементам)
`sqrt(x[, out])`	квадратный корень (для положительных) (по элементам)
`square(x [, out])`	квадрат (по элементам)
`reciprocal(x[, out])`	обратная величина (по элементам)

В.7. Тригонометрические функции для массивов

Все тригонометрические функции работают с радианами.

- x – массив
- out – массив нужного размера и формы для получения результата. Если

нет, то создаетя новый массив.

Функция	**Обратная функция**	**Описание**
`sin(x [, out])`	`arcsin(x [, out])`	синус (по элементам)
`cos(x [, out])`	`arccos(x [, out])`	косинус (по элементам)
`tan(x [, out])`	`arctan(x [, out])`	тангенс (по элементам)
	`arctan2(x1, x2 [, out])`	арктангенс x1/x2 с правильным выбором четверти (по элементам)
`hypot(x1, x2 [, out])`		гипотенуза по двум катетам (по элементам)
`sinh(x [, out])`	`arcsinh(x [, out])`	гиперболический синус (по элементам)
`cosh(x [, out])`	`arccosh(x [, out])`	гиперболический косинус (по элементам)
`tanh(x[, out])`	`arctanh(x[, out])`	гиперболический тангенс (по элементам)
`deg2rad(x[, out])`	`rad2deg(x[, out])`	преобразование градусов в радианы (по элементам)

B.8. Бинарные функции для массивов

- x1,x2 – массивы
- out – массив нужного размера и формы для получения результата. Если нет, то создаетя новый массив.

Функция	**Описание**
`add(x1, x2 [, out])`	сумма (по элементам)
`subtract(x1, x2 [, out])`	разность (по элементам)
`multiply(x1, x2 [, out])`	произведение (по элементам)
`divide(x1, x2 [, out])`	деление (по элементам)
`logaddexp(x1, x2 [, out])`	логарифм суммы экспонент (по элементам)
`logaddexp2(x1, x2 [, out])`	логарифм по основанию 2 от суммы экспонент (по элементам)

Продолжение на следующей странице

Таблица В.7 – *Продолжение*

Функция	**Описание**
`true_divide(x1, x2 [, out])`	истинное деление (с преобразованием типов)
`floor_divide(x1, x2 [, out])`	деление без преобразования типов (целочисленное)
`negative(x [, out])`	обратные элементы (по элементам)
`power(x1, x2 [, out])`	элементы первого массива в степени элементов из второго массива (по элементам)
`remainder(x1, x2 [, out]), mod(x1, x2 [, out]), fmod(x1, x2 [, out])`	остаток от деления (по элементам)
`greater(x1, x2 [, out])`	истина, если (x1 > x2) (по элементам)
`greater_equal(x1, x2 [, out])`	истина, если (x1 >= x2) (по элементам)
`less(x1, x2 [, out])`	истина, если (x1 < x2) (по элементам)
`less_equal(x1, x2 [, out])`	истина, если (x1 =< x2) (по элементам)
`not_equal(x1, x2 [, out])`	истина, если (x1 != x2) (по элементам)
`equal(x1, x2 [, out])`	истина, если (x1 == x2) (по элементам)
`logical_and(x1, x2 [, out])`	истина, если (x1 AND x2) (по элементам)
`logical_or(x1, x2 [, out])`	истина, если (x1 OR x2) (по элементам)
`logical_xor(x1, x2 [, out])`	истина, если (x1 XOR x2) (по элементам)
`logical_not(x [, out])`	истина, если (NOT x1) (по элементам)
`maximum(x1, x2 [, out])`	максимум из элементов двух массивов(по элементам)

В.9. Другие функции для массивов

- func1d – функция для вектора
- func – скалярная функция

- axis – индекс оси
- arr – массив
- *args, **kw – дополнительные аргументы
- nin – число входных параметров
- nout – число выходных параметров
- condlist - список условий
- funclist – список функций (для каждого условия)

Функция	**Описание**
`apply_along_axis(func1d, axis, a, *args)`	применить функцию к одномерному срезу вдоль оси
`apply_over_axes(func, a, axes)`	применить функцию последовательно вдоль осей
`vectorize(pyfunc [, otypes, doc])`	обобщить функцию на массивы
`frompyfunc(func, nin, nout)`	берет произвольную функцию Python и возвращает функцию Numpy
`piecewise(a, condlist, funclist, *args, **kw)`	применение кусочно-определенной функции к массиву

B.10. Сортировка, поиск, подсчет

- a – массив
- axis – индекс оси для сортировки (по умолчанию «–1» - последняя ось)
- kind – {‘quicksort’, ‘mergesort’, ‘heapsort’} тип сортировки
- order – индексы элементов, определяющие порядок сортировки
- keys – (k,N) массив из k элементов размера (N). k “колонок” будут отсортированы. Последний элемент – первичный ключ для сортировки.
- condition – матрица условий (маска)
- x, y – массивы для выбора элементов
- v – вектор
- side – {‘left’, ‘right’} позиция для вставки элемента (слева или справа от найденного индекса)

Функция	**Описание**
`sort(a [, axis, kind, order])`	отсортированная копия массива

Продолжение на следующей странице

Таблица В.9 – *Продолжение*

Функция	**Описание**
`lexsort(keys [, axis])`	Perform an indirect sort using a sequence of keys.
`argsort(a [, axis, kind, order])`	аргументы, которые упорядочивают массив
`array.sort([axis, kind, order])`	сортирует массив на месте (метод массива)
`msort(a)`	копия массива отсортированная по первой оси
`sort_complex(a)`	сортировка комплексного массива по действительной части, потом по мнимой
`argmax(a [, axis])`	индексы максимальных значений вдоль оси
`nanargmax(a [, axis])`	индексы максимальных значений вдоль оси (игнорируются NaN).
`argmin(a [, axis])`	индексы минимальных значений вдоль оси
`nanargmin(a [, axis])`	индексы минимальных значений вдоль оси (игнорируются NaN)
`argwhere(a)`	массив индексов ненулевых элементов. данные сгруппированы по элементам ([x1,y1,..], [x2,y2,..]....)
`nonzero(a)`	массивы индексов ненулевых элементов. сгруппированы по размерностям (индексы X, индексы Y, т.д.)
`flatnonzero(a)`	индексы ненулевых элементов в плоской версии массива
`where(condition, [x, y])`	возвращает массив составленный из элементов x (если выполнено условие) и y (в противном случае). Если задано только condition, то выдает его «не нули»
`searchsorted(a, v [, side])`	индексы мест, в которые нужно вставить элементы вектора для сохранения упорядоченности массива
`extract(condition, a)`	возвращает элементы (одномерный массив), по маске (condition)

Продолжение на следующей странице

Таблица В.9 – *Продолжение*

Функция	**Описание**
`count_nonzero(a)`	число ненулевых элементов в массиве

В.11. Статистика

- a – массив
- axis – индекс оси
- out – место для результата
- weights – веса
- returned – дополнительно выдать сумму весов
- dtype – тип данных для накопления суммы
- overwrite_input – использовать входящий массив для промежуточных вычислений
- ddof – дельта степеней свободы (см. описание)
- x,y – данные (строки – переменные, колонки - наблюдения)
- rowvar – если не 0, то строка переменные, колонки – наблюдения (если 0, то наоборот)
- bias – определеяет нормировку, совместно с ddof
- mode – {‘valid’, ‘same’, ‘full’} – объем выводимых данных
- old_behavior – совместимость со старой версией (без комплексного сопряжения)
- bins – разбиение по интервалам
- range – массив границ по x и по y
- normed – нормированное
- minlength – минимальное число интервалов при выводе

Функция	**Описание**
`amin(a [, axis, out])`	минимум в массиве или минимумы вдоль одной из осей
`amax(a [, axis, out])`	максимум в массиве или максимумы вдоль одной из осей
`nanmax(a [, axis])`	максимум в массиве или максимумы вдоль одной из осей (игнорируются NaN)
`nanmin(a [, axis])`	минимум в массиве или минимумы вдоль одной из осей (игнорируются NaN)

Продолжение на следующей странице

Таблица В.10 – *Продолжение*

Функция	**Описание**
`ptp(a [, axis, out])`	диапазон значений (максимум - минимум) вдоль оси
`average(a [, axis, weights, returned])`	взвешенное среднее вдоль оси
`mean(a [, axis, dtype, out])`	арифметическое среднее вдоль оси
`median(a [, axis, out, overwrite_input])`	вычисление медианы вдоль оси
`std(a [, axis, dtype, out, ddof])`	стандартное отклонение вдоль оси
`corrcoef(x [, y, rowvar, bias, ddof])`	коэффициенты корреляции
`correlate(a, v [, mode, old_behavior])`	кросс-корреляция двух одномерных последовательностей
`cov(m [, y, rowvar, bias, ddof])`	ковариационная матрица для данных
`histogram(a [, bins, range, normed, weights, ...])`	гистограмма из набора данных
`histogram2d(x, y [, bins, range, normed, weights])`	двумерная гистограмма для двух наборов данных
`histogramdd(sample [, bins, range, normed, ...])`	многомерная гистограмма для данных
`bincount(x [, weights, minlength])`	число появление значения в массиве неотрицательных значений
`digitize(x, bins)`	возвращает индексы интервалов к которым принадлежат элементы массива

В.12. Дискретное преобразование Фурье (numpy.fft)

- a – массив
- s – число элементов вдоль каждого направления преобразования (если больше размерности, то дополняются нулями)
- axes – последовательность осей для преобразования

- n – ширина окна
- d – шаг по частоте при выводе

Прямое преобразование	**Обратное преобразование**	**Описание**
`fft(a [, s, axis])`	`ifft(a [, s, axis])`	одномерное дискретное преобразование Фурье
`fft2(a [, s, axes])`	`ifft2(a [, s, axes])`	двумерное дискретное преобразование Фурье
`fftn(a [, s, axes])`	`ifftn(a [, s, axes])`	многомерное дискретное преобразование Фурье
`rfft(a [, s, axis])`	`irfft(a [, s, axis])`	одномерное дискретное преобразование Фурье (действительные числа)
`rfft2(a [, s, axes])`	`irfft2(a [, s, axes])`	двумерное дискретное преобразование Фурье (действительные числа)
`rfftn(a [, s, axes])`	`irfftn(a[, s, axes])`	многомерное дискретное преобразование Фурье (действительные числа)
`hfft(a [, s, axis])`	`ihfft(a [, s, axis])`	преобразование Фурье сигнала с Эрмитовым спектром
`fftfreq(n [, d])`		частоты дискретного преобразования Фурье
`fftshift(a [, axes])`	`ifftshift(a [, axes])`	преобразование Фурье со сдвигом нулевой компоненты в центр спектра

B.13. Линейная алгебра (numpy.linalg)

- a, b – матрицы
- out – место для результата
- ord – определяет способ вычисления нормы
- axes – массив осей для суммирования
- axis – индекс оси
- subscripts – индексы для суммирования
- *operands – список массивов
- dtype – тип результата
- offset – положение диагонали

- mode – {‘full’, ‘r’, ‘economic’} – выбор алгоритма разложения
- full_matrices – составлять полные матрицы
- compute_uv – выводить все матрицы
- rcond – граница для отбрасывания маленьких собственных значений
- ind – число индексов для вычисления обратной
- UPLO – {‘L’, ‘U’} выбирает часть матрицы для работы

Функция	**Описание**
`dot(a, b [, out])`	скалярное произведение массивов
`vdot(a, b)`	векторное произведение векторов
`inner(a, b)`	внутреннее произведение массивов
`outer(a, b)`	внешнее произведение векторов
`tensordot(a, b [, axes])`	тензорное скалярное произведение вдоль оси (размерность больше 1)
`einsum(subscripts, *operands [, out, dtype, ...])`	суммирование Эйншнейна Evaluates the Einstein summation convention on the operands.
`linalg.matrix_power(M, n)`	возведение квадратной матрицы M в степень n
`kron(a, b)`	произведение Кронекера двух массивов
`linalg.norm(a [, ord])`	норма матрицы или вектора
`linalg.cond(a[, ord])`	число обусловленности матрицы
`linalg.det(a)`	определитель
`linalg.slogdet(a)`	знак и натуральный логарифм определителя
`trace(a [, offset, axis1, axis2, dtype, out])`	сумма элементов по диагонали
`linalg.cholesky(a)`	разложение Холецкого
`linalg.qr(a[, mode])`	разложение QR
`linalg.svd(a[, full_matrices, compute_uv])`	сингулярное разложение
`linalg.solve(a, b)`	решение линейного матричного уравнения или системы скалярных уравнений

Продолжение на следующей странице

Таблица В.12 – *Продолжение*

Функция	**Описание**
`linalg.tensorsolve(a, b [, axes])`	решение тензорного уравнения a x = b для x
`linalg.lstsq(a, b [, rcond])`	решение матричного уравнения методом наименьших квадратов
`linalg.inv(a)`	обратная матрица (для умножения)
`linalg.pinv(a [, rcond])`	псевдо-обратная матрица (Мура-Пенроуза)
`linalg.tensorinv(a [, ind])`	«обратный» к многомерному массиву
`linalg.eig(a)`	собственные значения и правые собственные вектора квадратной
`linalg.eigh(a [, UPLO])`	собственные значения и собственные вектора эрмитовой или симметричной матрицы
`linalg.eigvals(a)`	собственные значения произвольной матрицы
`linalg.eigvalsh(a [, UPLO])`	собственные значения эрмитовой или действительной симметричной матрицы

В.14. Случайные величины (numpy.random)

- size - число элементов по каждому измерению

Функция	**Описание**
`rand(d0, d1, ..., dn)`	набор случайных чисел заданной формы
`randn([d1, ..., dn])`	набор (или наборы) случайных чисел со стандартным нормальным распределением
`randint(low [, high, size])`	случайные целые числа от low (включая) до high (не включая)
`random_integers(low[, high, size])`	случайные целые числа между low и high (включая).
`random_sample([size])`	случайные рациональные числа из интервала [0.0, 1.0]
`bytes(length)`	случайные байты

Продолжение на следующей странице

Таблица В.13 – *Продолжение*

Функция	**Описание**
`shuffle(x)`	тасовка элементов последовательностина месте
`permutation(x)`	возвращает последовательность, переставленных случайным образом элементов
`seed([seed])`	перезапуск генератора случайных чисел
`beta(a, b [, size])`	числа с Бетта- распределением $f(x, \alpha, \beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$ $B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt$
`binomial(n, p [, size])`	числа с биномиальным распределением $P(N) = \left(\frac{n}{N}\right) p^N(1-p)^{n-N}$
`chisquare(df [, size])`	числа с распределением хи-квадрат $p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}$
`exponential([scale, size])`	числа с экспоненциальным распределением $f(x, \frac{1}{\beta}) = \frac{1}{\beta} exp(-\frac{x}{\beta})$
`lognormal([mean, sigma, size])`	числа с логарифмическим нормальным распределением
`multivariate_normal(mean, cov [, size])`	числа с мульти нормальным распределением (mean – одномерный массив средних значений; cov – двухмерный симметричный, полож. определенный массив (N, N) ковариаций
`normal([loc, scale, size])`	числа с нормальным распределением
`poisson([lam, size])`	числа с распределением Пуассона
`standard_gamma(shape [, size])`	числа с гамма- распределением
`standard_normal([size])`	числа со стандартным нормальным распределением (среднее=0, сигма=1)
`standard_t(df [, size])`	числа со стандартным распределением Стьюдента с df степенями свободы
`uniform([low, high, size])`	числа с равномерным распределением

В.15. Полиномы (numpy.polynomial)

- coef – массив коэффициентов в порядке увеличения
- domain – область определения проецируется на окно
- window – окно. Сдвигается и масштабируется до размера области определения

Типы полиномов	Описание
`Polynomial(coef [, domain, window])`	разложение по степеням «x»
`Chebyshev(coef [, domain, window])`	разложение по полиномам Чебышева
`Legendre(coef [, domain, window])`	разложение по полиномам Лежандра
`Hermite(coef [, domain, window])`	разложение по полиномам Эрмита
`HermiteE(coef [, domain, window])`	разложение по полиномам Эрмита_E
`Laguerre(coef [, domain, window])`	разложение по полиномам Лагерра

Методы для полиномов

- p – полином
- x, y – набор данных для аппроксимации
- deg – степень полинома
- domain – область определения
- rcond – относительное число обусловленности элементы матрицы интерполяции с собственными значениями меньшими данного будут отброшены.
- full – выдавать дополнительную информацию о качестве полинома
- w – веса точек
- window – окно
- roots – набор корней
- m – порядок производной (интеграла)
- k – константы интегрирования
- lbnd – нижняя граница интервала интегрирования
- n – число точек разбиения
- size – число ненулевых коэффициентов

Методы полиномов	**Описание**
`__call__(z)`	полином можно вызвать как функцию
`convert([domain, kind, window])`	конвертирует в полином другого типа, с другим окном и т.д.
`copy()`	возвращает копию
`cutdeg(deg)`	обрезает полином до нужной степени
`degree()`	возвращает степень полинома
`deriv([m])`	вычисляет производную порядка m
`fit(x, y, deg [, domain, rcond, full, w, window])`	формирует полиномиальную интерполяцию степени deg для данных (x,y) по методу наименьших квадратов
`fromroots(roots [, domain, window])`	формирует полином по заданным корням
`has_samecoef(p)`	проверка на равенство коэффициентов
`has_samedomain(p)`	проверка на равенство области определения
`has_samewindow(p)`	проверка на равенство окна
`integ([m, k, lbnd])`	интегрирование
`linspace([n, domain])`	возвращает x,y - значения на равномерной сетке по области определения
`mapparams()`	возвращает коэффициенты масштабирования
`roots()`	список корней
`trim([tol])`	отбрасывает слагаемые с коэффициентами меньшими tole
`truncate(size)`	ограничивает ряд по количеству коэффициентов

C. Краткий справочник Matplotlib

C.1. Некоторые аргументы pyplot.plot

Аргумент	**Описание**
alpha	прозрачность $\in [0, 1)$
color или c	цвет линии
label	легенда
linestyle или ls	стиль линии ['-' | '–' | '-.' | ':' | 'steps' | ...]
linewidth или lw	ширина линии в пунктах
marker	вид маркера точки ['+' | ',' | '.' | '1' | '2' | '3' | '4']
markeredgecolor или mec	цвет края маркера
markeredgewidth или mew	толщина края маркера
markerfacecolor или mfc	цвет заливки маркера
markersize или ms	размер маркера

C.2. Некоторые функции интерфейса pyplot

Аргументы функций см. в документации к интерфейсу.

Функция	**Описание**
annotate	создание аннотаций на графике
arrow	стрелки на графике
axis	изменение пределов осей
axvline	вертикальная линия
bar	столбчатая диаграмма
cla	очистить график
draw	перестроить график
errorbar	график с ошибками
grid	координатная сетка

Продолжение на следующей странице

Таблица С.2 – *Продолжение*

Функция	**Описание**
`hist`	гистограмма
`imshow`	рисунок
`legend`	добавление легенды
`loglog`	график в логарифмическом масштабе по обеим осям
`pie`	секторная диаграмма
`polar`	график в полярных координатах
`quiver`	векторное поле
`savefig`	сохранение графика в файл