Н.Г. Михеев

Технологии для решения прикладных задач с использованием языка Python

Москва
Физический факультет МГУ им. М. В. Ломоносова
2020

Михеев Никита Глебович

Технологии для решения прикладных задач с использованием языка Python. –

М.: Физический факультет МГУ им. М. В. Ломоносова, 2020. – 100 с.

Данное пособие призвано показать возможности языка программирования Python для решения прикладных задач в технических и естественнонаучных областях. Предполагается, что читатель уже знаком с основами языка Python и желает освоить более продвинутые инструменты автоматизации, строящиеся на базе данного языка. В пособии рассматривается несколько наиболее популярных пакетов и технологий, используемых для создания простых прикладных программ, предназначенных для обработки и визуализации данных. Каждая глава снабжена примерами программного кода. Также имеются практические задания для самостоятельного выполнения.

Пособие рассчитано на студентов естественно-научных специальностей, но может быть полезно для всех, кто изучает или использует язык программирования Python для решения прикладных задач.

Подписано в печать _____. Тираж 30 экз. Заказ № ____
Физический факультет МГУ имени М.В. Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д. 1, стр. 2.

Оглавление

1 Программное обеспечение в формате интерактивных тетрадей

Python — это интерпретируемый язык программирования. Интерпретатор Python может работать как в режиме выполнения сценариев, так и в интерактивном режиме. В первом случае при запуске требуется указать файл с программным кодом. Во втором случае интерпретатор будет последовательно выполнять инструкции, вводимые пользователем через интерфейс командной строки.

Для разработки программ на Python не требуется каких-либо специальных сред разработки. Для написания программного кода подойдёт любой текстовый редактор. Тем не менее, существует достаточно много сред разработки, повышающих удобство редактирования и отладки сложных программ. В большинстве случаев по своим возможностям они аналогичны средами разработки для других языков программирования.

В последнее время в области научных вычислений набирают популярность программы на языке Python в формате интерактивных тетрадей. Интерактивные тетради подразумевают выполнение программ блоками инструкций, а также подробное документирование таких блоков. Для редактирования и выполнения интерактивных тетрадей требуется специальная программная оболочка. Одним из наиболее популярных средств является программное обеспечение, разрабатываемое в рамках проекта Jupyter.[1]

1.1 Тетради Jupyter Notebook

Jupyter Notebook — это средство для просмотра, редактирования и выполнения кода интерактивных тетрадей для научных вычислений. Тетради Jupyter в каком-то смысле аналогичны обычным бумажным тетрадям: в них можно размещать текст, формулы, изображения и производить вычисления. Тетради позволяют совмещать в одном документе программный код, его описание и результаты вычислений.

Среди примеров использования тетрадей Jupyter можно выделить, например, выложенные в открытый доступ тетради с обработкой экспериментальных данных детектора гравитационных волн LIGO.[2]

[1] https://jupyter.org

[2] https://mybinder.org/v2/gh/losc-tutorial/LOSC_Event_tutorial/master

Ссылки на другие интересные примеры тетрадей, выложенных в открытый доступ, можно найти в репозитории Jupyter на GitHub.[3]

1.2 Установка и запуск

Если Вы устанавливали Python в составе дистрибутива Anaconda, Jupyter у Вас уже есть. Если же Вы устанавливали Python как-то иначе, и Jupyter у Вас не установлен, его можно установить с помощью пакетного менеджера pip:

```
python -m pip install jupyter
```

Запуск сервера Jupyter осуществляется с помощью команды

```
jupyter notebook
```

При запуске сервера открывается браузер с обозревателем файлов (Рис. 1). Через браузер можно создавать и редактировать тетради в текущем каталоге и во вложенных в него каталогах. Для создания новой тетради нужно нажать кнопку *New* и выбрать язык программирования. Скорее всего, это будет Python 3 (Рис. 2). При создании новой тетради в текущем каталоге создаётся файл с расширением `.ipynb`, в котором будут храниться все данные тетради.

Рис. 1: Обозреватель файлов сервера Jupyter

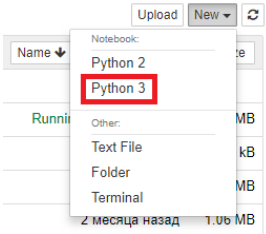


Рис. 2: Диалог выбора языка программирования при создании новой тетради

[3] https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks

1.3 Работа с ячейками

Тетрадь состоит из ячеек. Ячейки бывают нескольких видов:

- *Code* — исходный код программы, который будет исполняться при выполнении ячейки.
- *Markdown* — текстовое описание (комментарии) с разметкой на языке Markdown.
- *Raw NBConvert* — ячейки, содержимое которых не будет обрабатываться в Jupyter. Такие ячейки используются при экспорте тетрадей в другие форматы.
- *Heading* — устаревший тип ячеек. Вместо них рекомендуется использовать Markdown.

Запускать исполнение ячеек с кодом можно в следующих режимах: все сразу (пункт меню *Cell* > *Run All*), все над текущей (*Cell* > *Run All Above*), все под текущей (*Cell* > *Run All Below*), только текущую (*Run* или сочетание клавиш *Shift* + *Enter*). Последний вариант используется наиболее часто.

Выполнение ячейки с кодом — это выполнение инструкций, находящихся в данной ячейке. Переменные, созданные и инициализированные в ячейке, будут доступны из любых других ячеек. Запускать ячейки можно в произвольном порядке. Если ячейки используют общие данные, то порядок их выполнения может влиять на результат. После редактирования и повторного запуска ячеек или всей тетради контекст будет сохранён (в частности, сохранятся значения всех переменных). Это может привести к ошибкам, связанным с использованием значений переменных, заданных во время предыдущего запуска. Поэтому рекомендуется периодически перезагружать интерпретатор (*Kernel* > *Restart*), чтобы гарантированно удалить из памяти все возникшие в ходе работы объекты.

Среди прочего с ячейками можно совершать следующие операции: копирование, вставка, перемещение вверх и вниз, объединение, разделение на несколько ячеек.

1.4 Автоматическое дополнение и подсказки

В Jupyter есть автоматическое дополнение, которое позволяет быстро осуществлять подстановку имён, доступных из данной ячейки. Чтобы воспользоваться автоматическим дополнением, начните набирать имя переменной или функции и нажмите клавишу *Tab* — появится выпадающий список с вариантами дополнения (Рис. 3).

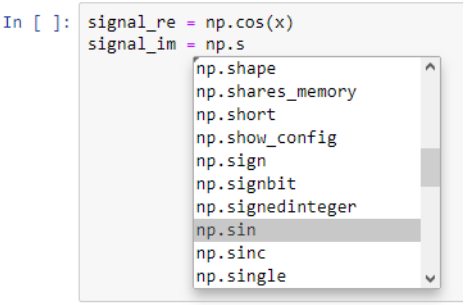


Рис. 3: Пример автоматического дополнения, вызываемого при нажатии на клавишу Tab

В Jupyter также есть возможность просмотра справочной информации о различных функциях (Рис. 4). Для получения справки, нужно поместить курсор на интересующую Вас функцию и набрать *Shift* + *Tab*. Справочная информация берётся из строк документации (docstring). Чтобы Jupyter показывал информацию о функциях из Вашего кода, после названий функций нужно помещать комментарии в тройных кавычках с текстом справки. Пример:

```
def test_jupyter_help():
    """
    Функция для тестирования справки Jupyter.
    """
    return
```

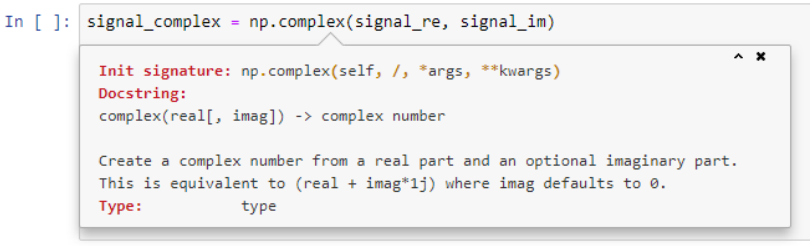


Рис. 4: Пример вывода справочной информации по функции

1.5 Markdown-форматирование

В ячейках типа «Markdown» можно писать текст, оформленный в соответствии с языком разметки GitHub Flavored Markdown. Описание синтаксиса и примеры использования можно найти в руко-

водстве GitHub.[4] Среди прочего в тексте можно использовать математические выражения, описанные в формате LaTeX. Если формула встраивается в текст, она помещается между одинарных символов $. Если выражение большое, и его нужно разместить отдельно, оно заключается между $$.

При выполнении ячеек с Markdown разметкой Jupyter обрабатывает текстовое описание и производит отрисовку документа. Пример ячейки с документацией в формате Markdown:

```
# Первый замечательный предел

$$\lim_{x \to 0}{\frac{\sin{x}}{x}} = 1$$

**Формулировка**: предел отношения синуса
к его аргументу при стремлении аргумента
к нулю равен единице.

Первый замечательный предел представляет собой
*устранимый разрыв*.
```

Вид ячейки после отрисовки приведённого выше кода изображён на рисунке 5.

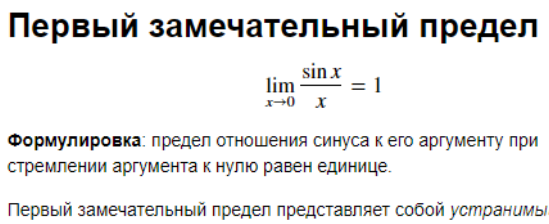


Рис. 5: Пример отрисованной ячейки тетради Jupyter с документацией в формате Markdown

1.6 Экспорт данных

Тетрадь Jupyter можно преобразовать в другие форматы. Это можно сделать, перейдя в меню *File* > *Download as*. Наиболее полезными являются следующие варианты:

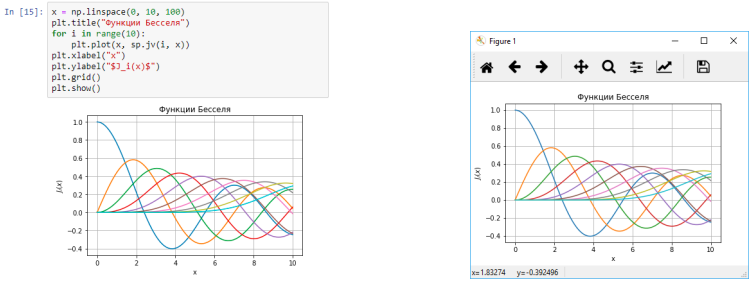
- Python (.py) — все блоки кода будут объединены в один сценарий на языке Python, Markdown-блоки будут добавлены в код в виде комментариев.

[4] https://guides.github.com/features/mastering-markdown

- HTML (.html) — тетрадь будет сохранена в виде статической HTML-страницы. Выполнять код в такой тетради нельзя. Зато для просмотра документа достаточно обычного браузера. Этот вариант удобно использовать, когда требуется отправить тетрадь пользователю, у которого может не быть программного обеспечения для просмотра тетрадей в формате .ipynb.

1.7 Встраивание диаграмм

По умолчанию диаграммы Matplotlib (и других пакетов) встраиваются в тетрадь в виде изображений (Рис. 6a). Для включения классического режим отображения графиков в отдельном окне (Рис. 6b) с возможностью интерактивного управления отображением (масштабирование, перемещение, сохранение...) нужно выполнить инструкцию `%matplotlib qt` (пишется отдельной строкой перед кодом вывода графика). Инструкция выполняется один раз и применяется ко всем графикам, которые будут строиться после её выполнения. Обратное переключение в режим встраивания графиков осуществляется с помощью инструкции `%matplotlib inline`.



(a) Встраивание в тетрадь Jupyter (b) В отдельном окне

Рис. 6: Примеры вывода диаграмм Matplotlib

1.8 Элементы пользовательского интерфейса

Пакет ipywidgets позволяет встраивать в тетради элементы пользовательского интерфейса. Ниже приведён пример, демонстрирующий возможность изменения параметра функции на графике с помощью ползунка и динамического перестроения графика при изменении параметра.

```
import ipywidgets as ipw

def plot_sin(freq):
    t = np.linspace(0, 4, 100)
    sin_signal = np.sin(t * 2 * np.pi * freq)
    plt.title("y = sin(2$\pi$fx); f = {}".format(freq))
    plt.plot(t, sin_signal)
    plt.show()

fs = ipw.FloatSlider(min=0, max=2, step=0.1, value=1.)
_ = ipw.interact(plot_sin, freq = fs)
```

Результат выполнения данного кода изображён на рисунке 7.

Для обеспечения отзывчивости интерфейса желательно выбирать шаг (step) не слишком маленьким. В противном случае обновление графика может заметно отставать от изменения параметра.

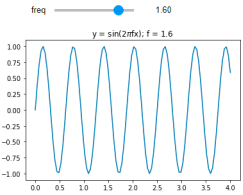


Рис. 7: Пример встраивания элементов пользовательского интерфейса в тетрадь Jupyter

Помимо ползунков в тетради можно встраивать: кнопки, поля для ввода текста, выпадающие списки, цветовые палитры, видео и аудио плееры, а также многое другое. Более подробную информацию о добавлении элементов графического интерфейса пользователя можно найти в описании библиотеки [3].[5]

1.9 Измерение производительности

В Jupyter предусмотрены команды, позволяющие оценить производительность кода. Для определения среднего времени выполнения функции используется инструкция `%timeit`. Команда вписывается перед вызовом функции в той же строке. Jupyter производит измерение времени выполнения серии вызовов указанной функции, вычисляет среднее время выполнения одного вызова и погрешность его определения. Пример использования:

```
def invert_matrix(m):
    return np.linalg.inv(m)

random_matrix = np.random.rand(100, 100)
%timeit invert_matrix(random_matrix)
```

[5] https://ipywidgets.readthedocs.io/en/stable/examples/Using%20Interact.html

```
399 μs ± 17.1 μs per loop
(mean ± std. dev. of 7 runs, 1000 loops each)
```

Для измерения времени выполнения текущей ячейки можно поместить вызов этой инструкции с дополнительным знаком % в начало ячейки: `%%timeit`.

Для детального анализа производительности функций можно использовать инструкцию `%prun`, которая позволяет определить времена выполнения всех вложенных функций.

1.10 Другие средства для работы с тетрадями

Помимо Jupyter Notebook в рамках проекта Jupyter выпускаются и другие инструменты для работы с интерактивными тетрадями. В частности, JupyterLab и JupyterHub.

JupyterLab — более новая и продвинутая среда для работы с интерактивными тетрадями. JupyterLab позволяет одновременно размещать в рабочей области сразу несколько документов, обозреватель файлов, консоль и другие инструменты. Более подробную информацию о JupyterLab можно получить из официальной документации.[6]

JupyterHub[7] — это серверное приложение, которое позволяет организовать одновременную работу с тетрадями для нескольких пользователей на удалённой машине.

1.11 Задание для самостоятельного выполнения

Провести исследование зависимости временных характеристик одного из алгоритмов сортировки от размера сортируемого массива. Оформить проведённое исследование в интерактивной тетради Jupyter.

Основные этапы работы

1. Проверить, что ПО Jupyter Notebook установлено. При необходимости установить.
2. Создать новую интерактивную тетрадь для экспериментов.

[6] https://jupyterlab.readthedocs.io/en/stable/
[7] https://jupyter.org/hub

3. Написать собственную реализацию одного из стандартных алгоритмов сортировки. Оформить алгоритм в виде функции в одной из ячеек тетради.
4. Сделать описание реализованного алгоритма и оформить его в формате Markdown. Желательно использовать максимальное количество элементов разметки (заголовки, списки, формулы, фрагменты кода...).
5. Провести измерение зависимости времени сортировки от количества элементов в массиве.
6. Построить график полученной зависимости и аппроксимировать зависимость функцией, которая предполагается моделью выбранного алгоритма. График исходной зависимости и её аппроксимации должен быть встроен в тетрадь.
7. Объяснить полученный результат. Оформить выводы в тетради.

2 Основы объектно-ориентированного программирования в Python

Код простой программы на Python может содержаться в одном файле исходного кода. С увеличением сложности программ часто используемые функции можно и нужно выносить в отдельные библиотеки. Это позволит избежать дублирования в коде. Кроме того распределение кода между несколькими файлами повышает удобство работы. Для создания программ, состоящих из нескольких файлов, в языке Python предусмотрен механизм модулей.

При разработке больших программ бывает удобно использовать подходы объектно-ориентированного программирования. Подобно другим языкам высокого уровня, Python позволяет объявлять в коде классы, содержащие поля и методы, а затем создавать и использовать экземпляры объектов объявленных классов. В этом разделе будут рассмотрены примеры создания и использования простых модулей, а также механизмы работы с классами.

2.1 Модули и способы их подключения

Модулем является любой файл, имеющий расширение `.py`. Чтобы получить доступ к функциям и классам модуля, его нужно подключить (импортировать) с помощью инструкции `import`. После

ключевого слова `import` следует имя модуля. Именем модуля является название файла без расширения. После импорта все объекты и функции, определённые внутри файла, будут доступны через оператор «`.`». Пример модуля и его использования из другого файла:

```
# Файл mymodule.py
def print_hello():
    print("Hello!")

def sum_a_and_b(a, b):
    return a + b
```

```
# Файл main.py
import mymodule

mymodule.print_hello()
c = mymodule.sum_a_and_b(3, 5)
```

Если имя модуля слишком длинное, при импорте для него можно назначить псевдоним (импортировать модуль под другим именем, которое отличается от оригинального). Для этого используется конструкция вида `import <модуль> as <псевдоним>`. В этом случае `main.py` из предыдущего примера может иметь вид:

```
import mymodule as mm

mm.print_hello()
c = mm.sum_a_and_b(3, 5)
```

С помощью конструкции `from <модуль> import <объект>` можно импортировать отдельные объекты в пространство имён текущего модуля. В этом случае использовать имя модуля и оператор «`.`» не придётся. Можно импортировать несколько объектов, перечислив их через запятую. Пример:

```
from mymodule import print_hello, sum_a_and_b

print_hello()
c = sum_a_and_b(3, 5)
```

Если в модуле есть код, находящийся за переделами функций и/или классов, при импорте этот код будет выполнен. Пример:

```
# Файл mymodule.py
def sum_a_and_b(a, b):
    return a + b

print("Test sum_a_and_b()")
print("1 + 3 =", sum_a_and_b(1, 3))
```

```
# Файл main.py
import mymodule
```

Консольный вывод при запуске файла `main.py`:

```
\>python main.py
Test sum_a_and_b()
1 + 3 = 4
```

Если требуется обеспечить выполнение кода при явном запуске файла модуля, но избежать выполнения кода при его импорте из других файлов, можно воспользоваться конструкцией `if __name__ == "__main__":`, которая проверяет имя модуля в текущем контексте. Модуль из предыдущего пример можно модифицировать следующим образом:

```
def sum_a_and_b(a, b):
    return a + b

if __name__ == "__main__":
    print("Test sum_a_and_b()")
    print("1 + 3 =", sum_a_and_b(1, 3))
```

Примеры запуска:

```
\>python main.py
\>python mymodule.py
Test sum_a_and_b()
1 + 3 = 4
```

По умолчанию Python ищет модули в текущей директории. Если в текущей директории требуемый модуль не обнаружен, производится обход директорий, находящихся в переменной `sys.path`. Ещё больше информации о работе с модулями можно найти в официальной документации.[8]

Если Вы пишете библиотеку, код которой распределён между несколькими файлами, разумно оформить её в виде пакета. *Пакет* – это каталог, внутри которого расположен файл с именем `__init__.py`. В пакете могут быть и другие файлы, но доступ к публичным объектам осуществляется именно через `__init__.py`. Пакеты импортируются аналогично модулям. Правильно оформленный пакет можно распространять через публичные репозитории и устанавливать с помощью пакетного менеджера pip.[9]

[8] https://docs.python.org/3/tutorial/modules.html
[9] https://packaging.python.org/tutorials/packaging-projects/

2.2 Классы

Для описания класса в языке Python используется ключевое слово `class`. За ключевым словом `class` следует имя класса и двоеточие. Описание класса размещается с отступом на строках ниже. Поля можно создать и инициализировать простым присвоением. Методы объявляются как обычные функции с использованием ключевого слова `def`.

В качестве первого аргумента во все методы автоматически передаётся ссылка на вызывающий их объект (его принято называть `self`). Поэтому все методы должны принимать как минимум один аргумент. При вызове метода ссылка на объект передаётся автоматически. Поэтому при вызове методов в них нужно передавать количество аргументов, которое будет на один меньше, чем количество, указанное при объявлении. Доступ из методов к полям и другим методам данного класса осуществляется через ссылку на текущий экземпляр (`self.<имя поля/метода>`).

Имена классов в Python принято записывать в `CamelCase` (слова пишутся с прописной буквы без пробела) [4]. Имена полей и методов — в `snake_case` (все слова начинаются со строчной буквы и разделяются подчёркиваниями). Пример объявления класса:

```
class MyTestClass:
    """
    Это класс для тестов
    """
    test_field = 3

    def print_hello(self):
        print("Hello")

    def is_equal_to_my_test_field(self, num):
        if num == self.test_field:
            return True
        return False
```

Для создания экземпляра класса, нужно вызвать функцию, название которой совпадает с именем класса. Для доступа к полям и методам используется оператор «`.`». Пример:

```
mc = MyTestClass()
mc.print_hello()
mc.test_field = 42
if mc.is_equal_to_my_test_field(42):
    print("mc.test_field is", mc.test_field)
```

Внутри класса можно объявить функции инициализаторы и финализаторы, которые в каком-то смысле аналогичны конструкторам и деструкторам в языке C++.

Инициализатор — метод, который автоматически выполняется при создании экземпляра класса. Для определения инициализатора внутри класса нужно создать метод с именем `__init__()`.

Финализатор — метод, который автоматически выполняется при удалении экземпляра класса. Для определения финализатора внутри класса нужно создать метод с именем `__del__()`.

В большинстве случаев инициализаторы и финализаторы используются для корректного выделения/удаления ресурсов, считывания/сохранения настроек, подключения/отключения оборудования. Пример объявления класса с инициализатором и финализатором:

```
class MyTestClass:
    def __init__(self):
        print("Объект создан")

    def __del__(self):
        print("Объект удалён")

mc = MyTestClass()
del mc
```

Результат выполнения:

```
Объект создан
Объект удалён
```

Часто вместо методов `__init__()`/`__del__()` соответствующий функционал определяется в методах `__enter__()`/`__exit__()`, которые могут использоваться вместе с менеджерами контекста.

Бывает полезно определить методы `__str__()` и `__repr__()`, которые возвращают строковые представления объекта. Оно требуется, например, при печати объекта функцией `print()`.

Больше информации о создании классов можно найти в официальной документации.[10]

[10] https://docs.python.org/3/tutorial/classes.html

2.3 Инкапсуляция

Инкапсуляция — это концепция скрытия внутренней реализации объекта от других компонентов. В частности, она проявляется в разделении методов и полей на приватные (private) и публичные (public) с целью ограничения доступа к служебным данным и функциям. В Python есть следующие уровни доступа к элементам модуля или класса:

- *Публичные* поля и методы — начинаются со строчной буквы.
- *Защищённые* поля и методы — начинаются с подчёркивания (_). Эти поля и методы доступны отовсюду. Наличие подчёркивания показывает, что это служебные элементы, и при нормальных условиях они не должны использоваться внешними ресурсами (но технически это не запрещено).
- *Приватные* поля и методы — начинаются с двойного подчёркивания (__). Такие поля и методы недоступны извне.

Пример использования полей с различными уровнями доступа:

```
class MyTestClass:
    my_public_field = 10
    _my_protected_field = 20
    __my_private_field = 30

mc = MyTestClass()

print("my_public_field =",
      mc.my_public_field)

print("_my_protected_field =",
      mc._my_protected_field)

# Обращение к приватному полю вызывает ошибку
try:
    print("__my_private_field =",
          mc.__my_private_field)
except AttributeError as e:
    print("Ошибка:\n" + str(e))
```

Результат выполнения:

```
my_public_field = 10
_my_protected_field = 20
Ошибка:
'MyTestClass' object has no attribute '__my_private_field'
```

2.4 Наследование

Механизм *наследования* позволяет передать (унаследовать) новому классу свойства и функции другого, уже определённого класса. Это позволяет избежать дублирования кода. Вместо того чтобы писать несколько реализаций одних и тех же методов для нескольких классов, можно создать *базовый класс*, в котором будут определены общие методы и поля, а в классах-наследниках определить только те поля и методы, которые являются уникальными и не должны присутствовать в других классах.

Чтобы определить класс-наследник, после имени определяемого класса в круглых скобках нужно перечислить имена базовых классов. Методы и поля базовых классов будут автоматически включены в описываемый класс. Пример наследования классов:

```
class Figure:
    center_x = 0
    center_y = 0

    def print_center(self):
        print("X = {}, Y = {}".format(self.center_x,
                                      self.center_y))

class Circle(Figure):
    radius = 5

    def print_radius(self):
        print("Radius = {}".format(self.radius))

c = Circle()
c.print_center()
c.print_radius()
```

2.5 Полиморфизм

Полиморфизм — это механизм, позволяющий создавать несколько реализаций одного и того же метода для обработки различных типов данных. Полиморфизм обеспечивает общий интерфейс для функций, которые похожи по своей сути, но различаются по реализации для различных типов данных.

Полиморфизм часто используется не только в программировании, но и в математике. Большинство операций определено для целого ряда математических объектов. Алгоритм выполнения этих операций может меняться в зависимости от конкретного типа объ-

екта. Тем не менее, эти операции имеют одинаковое название, одинаковую запись и похожий набор свойств. Например, произведение комплексных чисел подразумевает более сложный по сравнению с действительными числами алгоритм вычислений, но с точки зрения записи и свойств полностью совместимо с произведением действительных чисел.

Полиморфизм в языке Python реализуется путём переопределения методов для соответствующих классов. Пример:

```
class Figure:
    center_x = 0
    center_y = 0

    def print_info(self):
        print("Информация о фигуре:\n"\
              "  X = {}, Y = {}".format(self.center_x,
                                        self.center_y))

class Circle(Figure):
    radius = 5

    def print_info(self):
        print("Информация о круге:\n"\
              "  X = {}, Y = {}\n"\
              "  Radius = {}".format(self.center_x,
                                     self.center_y,
                                     self.radius))

f = Figure()
c = Circle()
f.print_info()
c.print_info()
```

Результат выполнения:

```
Информация о фигуре:
  X = 0, Y = 0
Информация о круге:
  X = 0, Y = 0
  Radius = 5
```

2.6 Задание для самостоятельного выполнения

1. Определить класс `EnhancedList`, являющийся наследником стандартного класса списка (`list`) и определить для него методы `length()` и `square()`.

   Метод `length()` должен возвращать количество элементов в

списке. Метод `square()` должен возводить в квадрат все элементы списка (пример: `[1, 2, 3]` -> `[1, 4, 9]`).

2. Написать сценарий для тестов. Сценарий и объявление класса должны находиться в разных файлах. Тестовый сценарий должен импортировать файл с определением класса, использовать два добавленных метода и как минимум один стандартный метод класса `list` (это позволит проверить, что при наследовании методы базового класса продолжают работать).

3 Основы работы с процессами в Python

Процесс по сути — это программа, запущенная в операционной системе. Процессы работают независимо друг от друга. У каждого процесса своё пространство виртуальной памяти. Без использования специальных средств один процесс не может получить доступ к памяти другого процесса.

Для работы с процессами из языка Python можно использовать модуль subprocess [2]. Одной из самых простых задач, которые решает данный модуль, является запуск сторонних программ. Пример запуска сторонней программы из программы на языке Python:

```
import subprocess
subprocess.call("notepad.exe")
```

В качестве аргумента в метод `call()` можно передать любую команду, которую можно выполнить в командной строке Windows или терминале Linux.

Сложные команды, которые могут иметь целый набор аргументов, удобно передавать в виде списка:

```
import subprocess

command = "ping"
url = "www.ya.ru"
iter_key = "-n"
num_iterations = str(5)
subprocess.call([command, url, iter_key, num_iterations])
```

`subprocess.call()` возвращает код завершения. Коды завершения определяются вызываемой программой (в данном случае `ping`). Если команда завершилась успешно, этот код обычно равен `0`, но бывают и исключения. Если при выполнении возникли ошибки, код завершения будет другим. Пример проверки кодов завершения:

```
import subprocess

result_code = subprocess.call("mkdir test_dir")
if result_code != 0:
    print("Ошибка!")
print("mkdir test_dir result:", result_code)

# Директория с таким именем уже существует,
# поэтому тут мы получим ошибку.
result_code = subprocess.call("mkdir test_dir")
if result_code != 0:
    print("Ошибка!")
print("mkdir test_dir result:", result_code)
```

Результат выполнения:

```
\> python .\subprocess_test.py
mkdir test_dir result: 0
mkdir: cannot create directory ‘test_dir’: File exists
Ошибка!
mkdir test_dir result: 1
```

Если требуется получить вывод команды для использования внутри скрипта, можно воспользоваться методом `check_output()`.

```
import subprocess

stdout = subprocess.check_output("ls")
file_names = stdout.decode("utf8").split("\n")

print("Файлы в текущем каталоге:")
for file_name in file_names:
    print(file_name)
```

При необходимости из программы на Python можно получить доступ не только к стандартному выводу, но и к стандартному вводу другого процесса. Для получения дальнейшей информации см. документацию к методу `subprocess.popen()`.[11]

4 Основы многопоточного программирования

Поток — это последовательность инструкций, которая может выполняться условно независимо от других таких же последовательностей инструкций в рамках процесса. Любая программа име-

[11] https://docs.python.org/3/library/subprocess.html

ет как минимум один поток. Внутри программы можно создать дополнительные потоки. В этом случае функции, запущенные в новых потоках, будут выполняться как бы параллельно с функциями из основного потока. В действительности параллельное выполнение потоков возможно только на многоядерных процессорах и далеко не во всех случаях, особенно в Python.[12]

Потоки в отличие от процессов делят между собой общие ресурсы. С одной стороны это повышает удобство обмена данными между потоками, с другой стороны вызывает целый ряд проблем, связанных с синхронизацией.

4.1 Запуск функций в отдельном потоке

Для работы с потоками можно использовать модуль threading. Чтобы запустить функцию в отдельном потоке, нужно сначала создать объект класса `threading.Thread()` с указанием функции, а затем запустить этот поток с помощью метода `start()`. Если при запуске в функцию нужно передать аргументы, они передаются в виде списка `args` при создании потока (`threading.Thread()`). Пример запуска функции в отдельном потоке:

```
import threading
import time

def do_job(job_name, delay):
    for i in range(3):
        print("Job: {}. Delay {} s. "\
              "Stage: {}".format(job_name, delay, i))
        time.sleep(delay)

t1 = threading.Thread(target=do_job, args=["T1 Job", 1.2])
t2 = threading.Thread(target=do_job, args=["T2 Job", 0.7])
t1.start()
t2.start()
```

Результат выполнения:

```
Job: T1 Job. Delay 1.2 s. Stage: 0
Job: T2 Job. Delay 0.7 s. Stage: 0
Job: T2 Job. Delay 0.7 s. Stage: 1
Job: T1 Job. Delay 1.2 s. Stage: 1
Job: T2 Job. Delay 0.7 s. Stage: 2
Job: T1 Job. Delay 1.2 s. Stage: 2
```

[12] https://habr.com/ru/post/84629/

Создаётся впечатление, что обе функции выполняются параллельно. На самом деле это не совсем так. Интерпретатор Python в любой момент времени в рамках одного процесса может выполнять не более одного потока. Поэтому программа на чистом Python, решающая определённую задачу в нескольких потоках, не будет работать быстрее, чем программа, решающая ту же задачу последовательно в одном потоке. Если же потоки взаимодействуют с внешними ресурсами и могут ждать от них ответа, использование многопоточности может значительно повысить производительность.

Иногда требуется получить ссылку на объект текущего потока (например, чтобы узнать его имя). Для этого можно использовать функцию `threading.current_thread()`.

Метод `threading.enumerate()` позволяет получить список ссылок на объекты всех активных на данный момент потоков.

Пример сбора информации о потоках:

```
import threading
import time

def print_thread_name():
    current_thread = threading.current_thread()
    print("Вызвано из потока {}".format(current_thread.name))
    time.sleep(1)

t1 = threading.Thread(target=print_thread_name, args=[])
t2 = threading.Thread(target=print_thread_name, args=[])
t1.start()
t2.start()

time.sleep(0.5)

threads = threading.enumerate()
print("Имеется {} активных потока:".format(len(threads)))
for thread in threads:
    print(" - ", thread.name)
```

Результат выполнения:

```
Вызвано из потока Thread-1
Вызвано из потока Thread-2
Имеется 3 активных потока:
 -  MainThread
 -  Thread-1
 -  Thread-2
```

4.2 Распределение задач между потоками посредством очередей

Если задачу можно разбить на несколько подзадач, подразумевающих взаимодействие с внешними ресурсами и ожидание от них каких-то результатов (операции с СУБД, сбор данных с измерительных устройство, запросы к внешним API), при запуске этих задач в отдельных потоках можно получить выигрыш в производительности. Выигрыш достигается за счёт того, что пока одни потоки ждут ответ от внешних ресурсов, другие потоки могут выполнять полезную работу, используя ресурсы локальной системы.

В случае, когда количество задач превышает количество потоков, распределять задачи между потоками можно несколькими способами. Самый простой способ — разделить все задачи поровну. Но если время выполнения задач сильно меняется в зависимости от задачи, такая стратегия является не совсем оптимальной. Некоторые потоки могут получить простые задачи и выполнить свою работу быстро, в то же время другим потокам достанутся более сложные задачи, и эти потоки будут работать дольше, затягивая общее время выполнения программы. В подобных случаях динамическое распределение (см. рисунок 8) является более эффективным. Такое распределение подразумевает выдачу задач потокам по мере их освобождения.

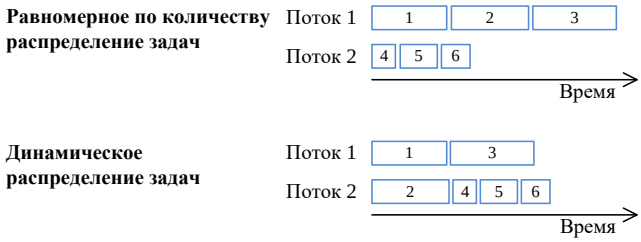


Рис. 8: Равномерное и динамическое распределение задач между потоками

Динамическое распределение задач можно реализовать посредством очередей (`Queue`). Данные помещаются в очередь с помощью метода `put()` с одной стороны и извлекаются из очереди с помощью метода `get()` с другой стороны (см. рисунок 9). При этом выполняется правило: то, что было добавлено в очередь раньше, раньше будет и извлечено из этой очереди.

Одновременно читать и писать данные в очередь могут сразу

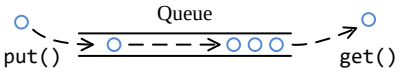


Рис. 9: Передача данных посредством очереди.

несколько потоков. Это и позволяет реализовать динамическое распределение задач. Пример организации динамического распределения задач посредством очередей:

```
import threading
import time
import queue

def process_item(input_queue):
    current_thread = threading.current_thread()

    while not input_queue.empty():
        item = input_queue.get()
        print(current_thread.name, "process item", item)
        time.sleep(1)

# Массив объектов, которые нужно обработать
items = [i for i in range(10)]

# Помещаем объекты в очередь
items_q = queue.Queue()
for item in items:
    items_q.put(item)

t1 = threading.Thread(target=process_item, args=[items_q])
t2 = threading.Thread(target=process_item, args=[items_q])
t1.start()
t2.start()
```

4.3 Синхронизация посредством Мьютексов

Часто требуется организовать *атомарное* (неразделимое, неразрывное) взаимодействие с различными объектами. Например, объект может содержать несколько взаимосвязанных полей. При одновременном обращении к такому объекту из разных потоков, комбинация полей объекта может оказаться некорректной, и это может привести к ошибкам. Пример подобной ситуации приведён ниже:

```
import threading
import math

class MyVector:
    x, y, z = (1, 1, 1)
    l = math.sqrt(3)  # Длина вектора

    def calculate_length(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

def process_item(v, offset):
    current_thread = threading.current_thread()

    for n in range(10000):
        l1 = v.l
        l2 = v.calculate_length()

        if abs(l1 - l2) > 0.001:
            print(current_thread.name,
                  "Неправильная длина вектора: "\
                  "{:.2} != {:.2}".format(l1, l2))

        v.x = (n + offset) % 10  # Изменяем вектор
        v.l = v.calculate_length()

v = MyVector()
t1 = threading.Thread(target=process_item, args=[v, 0])
t2 = threading.Thread(target=process_item, args=[v, 7])
t1.start()
t2.start()
```

Результат выполнения программы:

```
Thread-2 Неправильная длина вектора: 1.7 != 6.2
Thread-1 Неправильная длина вектора: 1.7 != 6.2
Thread-1 Неправильная длина вектора: 9.1 != 1.4
```

Из примера видно, что при большом количестве вызовов конечное (немгновенное) время обновления и чтения данных приводит к ошибкам. Чтобы избежать подобных ситуаций, нужно обеспечить атомарность чтения и записи данных. *Атомарные операции* — последовательность инструкций, которые должны выполняться друг за другом в строго определённом порядке. Добавление дополнительных инструкций (например, за счёт действий из других потоков) в атомарную последовательность не допускается.

Атомарные операции можно организовать с помощью *мьютексов* — объектов класса `threading.Lock()`.

Мьютексы (локи) работают следующим образом:

- Поток, из которого требуется получить доступ к общему ресурсу, вызывает метод `acquire()` объекта `Lock`. Если ресурс свободен (не захвачен другим потоком), то он захватывается.
- При попытке захвата мьютекса, который уже захвачен другим потоком, происходит приостановка выполнения кода на функции `acquire()`. Код продолжит выполняться после того, как ресурс будет освобожден.
- После того, как поток, захвативший мьютекс, выполнил все необходимые операции с общим ресурсом, он должен освободить ресурс путём вызова метода `release()`.
- После освобождения ресурса, у других потоков появляется возможность захватить его путём вызова метода `acquire()`.

Чтобы организовать безопасную работу с вектором в примере выше, в класс вектора нужно добавить мьютекс

```
class MyVector:
    x, y, z = (1, 1, 1)
    l = math.sqrt(3)  # Длина вектора
    lock = threading.Lock()
```

а в цикле, где производится модификация вектора, осуществлять блокировку и освобождение общего ресурса (вектора):

```
v.lock.acquire()
l1 = v.l
l2 = v.calculate_length()
v.lock.release()
```

Между захватом и возвратом могут возникнуть ошибки, приводящие к выбросу исключений. Поэтому желательно предусмотреть их обработку, а освобождение ресурса организовать внутри блока `finally`:

```
v.lock.acquire()
try:
    # Тут может возникнуть исключение
    v.x = (n + offset) % 10  # Изменяем вектор
    v.l = v.calculate_length()
finally:
    v.lock.release()
```

Чтобы избежать блокировок, которые могут возникнуть при выполнении метода `acquire()` без последующего вызова `release()`, можно использовать *менеджер контекста* (конструкция вида `with v.lock:`), который автоматически освободит ресурс при выходе из соответствующего блока кода.

4.4 События

Работу нескольких потоков можно синхронизировать посредством событий. *События* по сути — это флаги, которые гарантированно синхронизируются между потоками.

Чтобы управлять потоками посредством событий, нужно создать экземпляр класса `threading.Event()` и передать его в соответствующие потоки. Для установки и очистки события используются методы `set()` и `clear()`. Чтобы приостановить выполнение кода потока до возникновения какого-то события следует использовать метод `wait()`. Пример работы с потоками и событиями:

```
import threading
import time

def process_item(event, time_to_sleep):
    for i in range(3):
        t = time.strftime("%H:%M:%S", time.gmtime())
        print("{} Часть 1. Стадия {}.".format(t, i))
        time.sleep(time_to_sleep)

    print("Ожидание события...")
    event.wait()

    for i in range(3):
        t = time.strftime("%H:%M:%S", time.gmtime())
        print("{} Часть 2. Стадия {}.".format(t, i))
        time.sleep(time_to_sleep)

e = threading.Event()
e.clear()

t1 = threading.Thread(target=process_item, args=[e, 1])
t1.start()

# Ожидание выполнения первой части обработки
time.sleep(10)
e.set()
```

Результат выполнения:

```
15:30:15 Часть 1. Стадия 0.
15:30:16 Часть 1. Стадия 1.
15:30:17 Часть 1. Стадия 2.
Ожидание события...
15:30:25 Часть 2. Стадия 0.
15:30:26 Часть 2. Стадия 1.
15:30:27 Часть 2. Стадия 2.
```

4.5 Задание для самостоятельного выполнения

Написать многопоточную программу для измерения времени ответа сетевых узлов с динамическим распределением задач и сбором результатов посредством очередей.

Измерение времени ответа должно осуществляться путём однократного выполнения системной команды `ping` в отдельном процессе и последующего разбора консольного вывода этой команды.

Перед началом измерений в основном потоке нужно составить список внешних ресурсов для исследования. Пример:

```
sites = ["www.ya.ru",
         "www.google.com",
         "www.vk.com",
         "www.msu.ru"]
```

Все элементы этого списка нужно последовательно поместить в очередь входных данных. Задачи должны быть распределены минимум между двумя потоками.

Функция обработчик, вызываемая в отдельных потоках, должна брать имена сайтов из очереди, осуществлять измерения времён ответов и записывать результаты измерений во вторую очередь — очередь результатов измерений. В очередь с результатами должны сохраняться пары (адрес сайта, время ответа), поскольку в случае выполнения работы нескольким потоками очерёдность добавления результатов не определена.

В основном потоке требуется дождаться завершения работы всех потоков, осуществляющих измерения (можно воспользоваться функцией `threading.active_count()`), прочитать из очереди и вывести в консоль результаты всех измерений.

5 Символьные вычисления

Символьные вычисления — вид вычислений, которые подразумевают математические преобразования над обобщёнными переменными (символами). Данный подход отличается от численных расчётов, при которых все операции применяются непосредственно к числам (значениям переменных). Символьные вычисления выполняются аналитически, и только на последней стадии при необходимости производится подстановка конкретных значений чисел. Системы символьных вычислений могут работать не только с числами, но и с переменными, конкретные значения которых не за-

даны. Ответ при этом будет выдаваться в виде математического выражения с использованием этих переменных.

Для языка Python есть пакет SymPy [5], предназначенный для символьных вычислений. SymPy производит различные преобразования над символами (объектами класса `Symbol`), символьными функциями (объектами класса `Function`) и выражениями, состоящими из символов и символьных функций. Для символов переопределены базовые математические операции Python. Пример:

```
import sympy as sp
x = sp.Symbol("x")
print(x + x)
```

Результат выполнения:

```
2*x
```

Если среда разработки поддерживает отображение математических выражений в формате LaTeX (одной из таких сред является Jupyter Notebook), в SymPy можно включить вывод результатов в соответствующем формате следующим образом:

```
import sympy as sp
sp.init_printing(use_latex="mathjax")
```

Настройки вывода распространяются только на вывод интерпретатора и вывод с помощью функции `display()` (встроенная функция IPython/Jupyter). На функцию `print()` эти настройки не распространяются.

Примеры простых символьных выражений и их вывода в формате LaTeX:

```
x = sp.Symbol("x")
x
```

x

```
x+x
```

$2x$

```
1/x
```

$\frac{1}{x}$

```
x/(x+1)**2
```

$\frac{x}{(x+1)^2}$

Здесь и в последующих примерах предполагается, что импорт и настройка пакета SymPy уже выполнены.

При создании символьной переменной передаётся строка с отображаемым именем. Желательно, чтобы отображаемые имена символов совпадали с именами переменных.

Пример именования, которое вызывает путаницу. Так делать не нужно!

```
y = sp.Symbol("z")
y
```

z

Для именования символов можно использовать буквы греческого алфавита. В этом случае нужно передавать имя соответствующей буквы:

```
xi = sp.Symbol("xi")
```

Некоторые константы уже определены в SymPy в качестве символов: `sp.pi` — число π, `sp.E` — число e, `sp.oo` — ∞. Полный список констант приведён в официальной документации [5].

В случаях, когда требуется определить сразу несколько символов, удобно использовать функцию `symbols()`, которая принимает строку с именами символов, перечисленными через пробел, и возвращает кортеж символьных переменных:

```
x, y, z = sp.symbols("x y z")
```

Символьные функции являются объектами класса `Function`. Разница между символами и функциями проявляется, например, при решении дифференциальных уравнений. При использовании функций, нужно указывать их аргумент, который, в свою очередь, должен быть символьной переменной.

```
f = sp.Function("f")
x = sp.Symbol("x")
f(x)
```

$f(x)$

5.1 Базовые математические функции

Символы не являются числами. Поэтому к ним нельзя применять функции из таких пакетов, как Math или NumPy. Вместо них нужно использовать соответствующие символьные аналоги, определённые в пакете SymPy.

Примеры некоторых математических функций SymPy:

```
sp.sqrt(x+1)
```

$\sqrt{x+1}$

```
sp.exp(-x)
```

e^{-x}

Из символьных переменных и функций можно составлять символьные выражения и присваивать их переменным.

```
frac = x/(x + 1)
```

Над выражениями можно также осуществлять различные операции и использовать их в качестве составных частей других выражений:

```
frac**2 - sp.sqrt(frac)
```

$$\frac{x^2}{(x+1)^2} - \sqrt{\frac{x}{x+1}}$$

Функция `subs()` осуществляет подстановку числовых значений в символьное выражение. Данная функция принимает словарь, ключами которого являются символьные переменные, а значениями — числа, которые требуется подставить вместо соответствующих символов. В функцию `subs()` не обязательно передавать значения всех переменных. Допускаются частичные подстановки.

```
p = (x + y) / (x + 1)
p.subs({x : 1, y : 2})
```

$$\frac{3}{2}$$

После подстановки числовых значений SymPy часто возвращает результат не в виде одного числа, а в виде некоторого выражения (в примере выше таким выражением является натуральная дробь). Чтобы вычислить значение выражения и представить его в виде десятичной дроби, можно вызвать метод `n()`:

```
sp.pi.n()
```

```
3.14159265358979
```

Функция `factor()` позволяет представить многочлен в виде произведения:

```
x = sp.Symbol("x")
p = x**5 - 15 * x**3 + 10 * x**2 + 60 * x - 72
sp.factor(p)
```

$$(x-2)^3 (x+3)^2$$

С помощью функции `collect()` можно группировать степени:

```
x, y = sp.symbols("x y")
p = x**2 + x**2*y + y**3 + y + 1
sp.collect(p, y)
```

$$x^2 + y^3 + y\left(x^2 + 1\right) + 1$$

Раскрытие скобок осуществляется с помощью функции `expand()`:

```
x = sp.Symbol("x")
sp.expand((x + 1)**5)
```

$$x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$$

Функция `simplify()` упрощает математические выражения:

```
x = sp.Symbol("x")
sp.simplify(x**3 - (x - 1)**2 -2*x + x**2)
```

$x^3 - 1$

По умолчанию SymPy не приводит дроби к общему знаменателю. Чтобы привести дроби к общему знаменателю, следует использовать функцию `together()`.

```
x, y = sp.symbols("x y")
p = 1 / (x + 1) + 1 / (x + y)
p
```

$\frac{1}{x+y} + \frac{1}{x+1}$

```
sp.together(p)
```

$\frac{2x+y+1}{(x+1)(x+y)}$

5.2 Основы математического анализа

Для вычисления пределов используется функция `limit()`, принимающая 3 аргумента:

- Функция, предел которой требуется вычислить.
- Аргумент, который нужно устремить к заданной точке.
- Точка, в которой требуется вычислить предел функции.

```
x = sp.Symbol("x")
f = (1 + 1/x)**x
x_0 = sp.oo
sp.limit(f, x, x_0)
```

e

Для вычисления производных используется функция `diff()`. В качестве первого аргумента передаётся дифференцируемая функция. В качестве второго — переменная, по которой требуется произвести дифференцирование.

```
x = sp.Symbol("x")
p = x/sp.ln(x)
sp.diff(p, x)
```

$\frac{1}{\log(x)} - \frac{1}{\log^2(x)}$

Функцию `diff()` можно вызвать как собственный метод выражения или функции.

```
x, y = sp.symbols("x y")
p = y * x**2 + sp.ln(x + y**2)
p.diff(y)
```

$$x^2 + \frac{2y}{x+y^2}$$

```
f = sp.Function("f")
x = sp.Symbol("x")
(f(x)**2).diff(x)
```

$$2f(x)\frac{d}{dx}f(x)$$

Интегрирование в SymPy осуществляется с помощью функции `integrate()`. Для неопределённых интегралов синтаксис аналогичен функции `diff()`. При выводе результатов `integrate()` не выводит константу интегрирования. В будущих версиях SymPy константы интегрирования могут появиться.[13]

```
x = sp.Symbol("x")
p = 1/sp.sqrt(1 + x**2)
sp.integrate(p, x)
```

$$\operatorname{asinh}(x)$$

Более корректным ответом в данном случае будет $\operatorname{asinh}(x) + C$.

Для вычисления определённых интегралов вместо переменной интегрирования в `integrate()` нужно передать кортеж, содержащий переменную интегрирования, нижний и верхний пределы:

```
x = sp.Symbol("x")
p = 1/sp.sqrt(1 + x**2)
sp.integrate(p, (x, 0, 1))
```

$$\log\left(1 + \sqrt{2}\right)$$

Для разложения в ряд Тейлора используется функция `series()`. Аргументы:

- Переменная, по которой требуется провести разложение.
- Точка, в окрестности которой требуется разложить функцию.
- Количество членов ряда, которое требуется вычислить.

```
x = sp.Symbol("x")
sp.exp(x).series(x, 0, 5)
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \mathcal{O}\left(x^6\right)$$

Для суммирования рядов используется функция `summation()`. Аргументы:

- Выражение для n-го члена.
- Кортеж, включающий в себя:

[13] https://github.com/sympy/sympy/issues/11756

- Индекс, по которому производится суммирование.
- Начальное значение индекса суммирования.
- Конечное значение индекса суммирования.

```
n = sp.Symbol("n")
sp.summation(1/2**n, (n, 1, sp.oo))
```

```
1
```

```
x = sp.Symbol("x")
n = sp.Symbol("n")
sp.summation(x**n/sp.factorial(n), (n, 0, sp.oo))
```

$$e^x$$

5.3 Решение уравнений

Для решения алгебраических уравнений используется функция `solve()`. Функция принимает уравнение и переменную, относительно которой требуется решить уравнение. По умолчанию ищется решение уравнения $f(x) = 0$.

```
x = sp.Symbol("x")
f = x**3 - x**2 + x
sp.solve(f, x)
```

$$\left[0, \quad \frac{1}{2} - \frac{\sqrt{3}i}{2}, \quad \frac{1}{2} + \frac{\sqrt{3}i}{2}\right]$$

Уравнение с ненулевой правой частью можно задать с помощью функции `Eq()`:

```
x, a, b = sp.symbols("x a b")
sp.solve(sp.Eq(a*x, b), x)
```

$$\left[\frac{b}{a}\right]$$

Системы уравнений задаются в виде списков уравнений, входящих в систему. Переменные также передаются в виде списка:

```
x, y = sp.symbols("x y")
sp.solve([x + 1, x + y + 5], [x, y])
```

$$\{x : -1, \quad y : -4\}$$

Для решения дифференциальных уравнений используется функция `dsolve()`. Решением дифференциального уравнения является

функция. Поэтому символьное выражение, описывающее уравнение, помимо символов (объектов класса `Symbol`) должно содержать как минимум одну функцию (объект класса `Function`). В качестве первого аргумента `dsolve()` принимает уравнение. Второй аргумент — функция, относительно которой нужно решить уравнение.

Пример решения уравнения колебаний:

$\frac{d^2 f(x)}{dx^2} + f(x) = 0$

```
f = sp.Function("f")
x = sp.Symbol("x")
sp.dsolve(sp.Eq(f(x).diff(x, x) + f(x), 0), f(x))
```

$f(x) = C_1 \sin(x) + C_2 \cos(x)$

Для получения ответа с учётом начальных условий нужно дополнительно решить систему алгебраических уравнений.

Уравнение колебаний с начальными условиями:

$$\begin{cases} \frac{d^2 f(x)}{dx^2} + f(x) = 0 \\ f(0) = 0 \\ f(\pi/2) = 1 \end{cases}$$

```
f = sp.Function("f")
x = sp.Symbol("x")
# Общее решение дифференциального уравнения
gs = sp.dsolve(f(x).diff(x, x) + f(x), f(x))

# Общее решение представляется в виде равенства
# f(x) = f(x, C1, C2, ...)
# Нас интересует правая часть этого равенства
f = gs.args[1]

# Решаем систему алгебраических уравнений
# для начальных условий
# Условие 1: f(0) = 0
# Условие 2: f(pi/2) = 1
# Нужно решить систему относительно констант C1 и C2
c_val = sp.solve(
    [sp.Eq(f.subs({x:0}), 0),
     sp.Eq(f.subs({x:sp.pi/2}), 1)],
    ["C1", "C2"])

# Подставляем найденные значения констант C1 и C2
f.subs(c_val)
```

$\sin(x)$

5.4 Линейная алгебра

Матрица в SymPy представляется объектом класса `Matrix`. Матрицу можно определить, явно задав её элементы:

```
A = sp.Matrix([[1, 2, 0],
               [3, 1, 2],
               [1, 1, 0]])
A
```

$$\begin{bmatrix}1 & 2 & 0\\3 & 1 & 2\\1 & 1 & 0\end{bmatrix}$$

Также матрицы можно задавать с помощью функций, вычисляющих значения элементов по их индексам:

```
def f(i, j):
    if i == j:
        return 1
    else:
        return 0

sp.Matrix(4, 4, f)
```

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

Для матриц и векторов перегружен оператор умножения (`*`).

```
A = sp.Matrix([[1, 2, 0],
               [3, 1, 2],
               [1, 1, 0]])
x = sp.Matrix([1, 2, 3])
A * x
```

$$\begin{bmatrix}5\\11\\3\end{bmatrix}$$

Для вычисления определителей используется функция `det()`:

```
sp.det(A)
```

$$2$$

Вычисление обратной матрицы осуществляется с помощью метода `inv()`:

```
A.inv()
```

$$\begin{bmatrix}-1 & 0 & 2\\1 & 0 & -1\\1 & \frac{1}{2} & -\frac{5}{2}\end{bmatrix}$$

Метод `eigenvals()` возвращает словарь, ключами которого являются собственные значения, а значениями — их кратности.

```
A = sp.Matrix([[1, 0, 0],
               [1, 0, 1],
               [0, 1, 0]])
A.eigenvals()
```

$$\{-1 : 1,\quad 1 : 2\}$$

Метод `eigenvects()` возвращает список троек вида: собственное значение, кратность собственного значения, список собственных векторов, соответствующих данному собственному значению.

```
A.eigenvects()
```

$$\left[\left(-1,\quad 1,\quad \left[\begin{bmatrix}0\\-1\\1\end{bmatrix}\right]\right),\quad \left(1,\quad 2,\quad \left[\begin{bmatrix}0\\1\\1\end{bmatrix}\right]\right)\right]$$

5.5 Задание для самостоятельного выполнения

Оценить пригодность пакета SymPy для проверки решения базовых задач математического анализа.

Средствами пакета SymPy требуется получить аналитическое решение 9 задач из задачника В.Ф. Бутузов, Н.Ч. Крутицкая, Г.Н. Медведев, А.А. Шишкин. Математический анализ в вопросах и задачах. М.: ФИЗМАТЛИТ, 2000. При желании можно использовать любой другой задачник по математическому анализу.

Рекомендуется взять по 3 задачи из следующих разделов:

- Глава III. Предел функции. Непрерывность функции.
- Глава IV. Производные и дифференциалы.
- Глава V. Неопределённый интеграл.

Ответы, полученные средствами SymPy, сравнить с ответами, приведёнными в задачнике. Если ответы не совпадают, объяснить причины расхождения.

6 Работа с табличными данными

Обработка результатов экспериментов часто подразумевает работу с наборами данных, упорядоченных в виде таблиц. Типичными задачами являются вычисление средних значений, дисперсий, коэффициентов корреляции, различные массовые алгебраические преобразования строк или столбцов таблиц, построение графиков.

Для решения подобных задач часто используют редакторы электронных таблиц, такие как Microsoft Office Excel, iWork Numbers, Libre Office Calc... Но иногда такие задачи удобнее решать с использованием языков программирования, например в случаях когда:

- Обрабатываемые данные изначально являются результатом работы программы, взаимодействующей с оборудованием.
- Требуется нестандартная пред- или постобработка данных (например, фильтрация шумов, вычисление спектров, сложная визуализация, автоматическая публикация и подготовка к ней).

- Обработка данных подразумевает сложные математические преобразования, реализовать которые в редакторах таблиц сложно или невозможно.
- Требуется автоматизировать большое количество однотипных операций.

Удобным инструментом для решения подобных задач на языке Python является пакет pandas [6]. Этот пакет позволяет решать все перечисленные выше задачи и по своему функционалу похож на редактор электронных таблиц.

6.1 Основы работы с таблицами в pandas

Одним из основных объектов, с которыми работает pandas, является объект `DataFrame`. Этот объект представляет собой таблицу, содержащую значения (`values`). Таблица образована колонками (`columns`) и строками (`index`). По умолчанию строки и колонки индексируются числами, но для удобства работы можно присвоить им произвольные имена. Пример создания таблицы размером 4×3, заполненной случайными числами, с явно заданными именами строк и столбцов:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4, 3),
                  index=["L 1", "L 2", "L 3", "L 4"],
                  columns=["Col A", "Col B", "Col C"])
print(df)
```

Результат выполнения:

```
        Col A     Col B     Col C
L 1 -0.189277  1.380569 -0.080597
L 2 -0.611691 -1.561334  1.996562
L 3 -1.999348 -1.018972 -0.435398
L 4  0.221022 -1.705956  1.190567
```

Обратившись к полю `values`, можно получить данные в виде двумерного массива NumPy. В полях `columns` и `index` содержатся списки имён столбцов и строк. Размер таблицы хранится в поле `shape`. Пример вывода информации о таблице:

```
print("Table shape:", df.shape)
print("Columns:", df.columns)
print("Index:", df.index)
print("Values:\n", df.values)
```

Результат выполнения:

```
Table shape: (4, 3)
Columns: Index(['Col A', 'Col B', 'Col C'], dtype='object')
Index: Index(['L 1', 'L 2', 'L 3', 'L 4'], dtype='object')
Values:
 [[-0.18927749  1.3805687  -0.08059736]
 [-0.61169104 -1.56133396  1.99656243]
 [-1.99934831 -1.01897207 -0.43539816]
 [ 0.22102164 -1.70595569  1.19056656]]
```

6.2 Загрузка данных из файлов

Одним из наиболее популярных текстовых форматов для хранения табличных данных является формат CSV. Элементы таблиц в CSV отделяются друг от друга запятыми. В первой строке могут содержаться названия столбцов, а в первой колонке — названия строк. Пример таблицы в формате CSV:

```
h,t1,t2,t3
10,0.1,0.11,0.85
20,0.2,0.22,0.21
30,0.33,0.32,0.32
```

Следующий код позволяет загрузить в `DataFrame` таблицу, хранящуюся в файле `test_table.csv`.

```
data = pd.read_csv("test_table.csv")
```

Если вместо запятой используется другой разделитель (например, табуляция), его нужно передать явно с помощью аргумента `sep`. Пример чтения данных из файла в формате CSV с нестандартным разделителем:

```
data = pd.read_csv("table_tab_sep.csv", sep="\t")
```

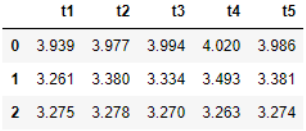
По умолчанию предполагается, что первая строка содержит имена столбцов, а имена строк в файле не указаны. Если файл не соответствует формату, используемому по умолчанию, формат можно поменять: указание колонки с именами строк осуществляется с помощью аргумента `index_col`, отключение заголовков столбцов осуществляется с помощью аргумента `header` (нужно передать `header=None`). Пример:

```
data = pd.read_csv("table.csv", index_col="Длина")
```

Pandas позволяет читать данные не только из файлов в формате CSV, но и из других форматов, среди которых XLSX, JSON, SQL.

6.3 Вывод таблиц

IPython (и в частности, Jupyter) поддерживает печать таблиц с HTML-форматированием (см. рисунок 10). HTML-форматирование используется при выводе таблиц интерпретатором, или при использовании функции `display()`. Функция `print()` всегда выводит таблицы в текстовом виде.



	t1	t2	t3	t4	t5
0	3.939	3.977	3.994	4.020	3.986
1	3.261	3.380	3.334	3.493	3.381
2	3.275	3.278	3.270	3.263	3.274

Рис. 10: Пример форматированного вывода таблиц

6.4 Обращение к элементам таблиц

Примеры, приведённые ниже, будут подразумевать использование следующей таблицы:

```
      c1    c2    c3
l1  0.52  0.83  0.97
l2  0.18  0.63  0.07
l3  0.94  0.39  0.49
l4  0.10  0.87  0.39
```

Осуществлять запись и чтение значений элементов с использованием имен строк и столбцов можно с помощью поля `at`. Пример:

```
v = df.at["l1", "c2"]
print(v)
df.at["l1", "c1"] = 0.33
print(df)
```

Результат выполнения:

```
0.83
      c1    c2    c3
l1  0.33  0.83  0.97
l2  0.18  0.63  0.07
l3  0.94  0.39  0.49
l4  0.10  0.87  0.39
```

Аналогичным образом можно обращаться к элементам таблицы по номерам их строк и столбцов. Для этого используется поле `iat`. Строки и столбцы нумеруются с нуля. Пример:

```
v = df.iat[0, 0]
print(v)
```

Результат выполнения:

```
0.33
```

Pandas позволяет читать и изменять не только отдельные элементы таблиц, но и целые столбцы, строки и прямоугольные диапазоны. Для этого используются поля `loc` (для обращения по именам) и `iloc` (для обращения по номерам). Задание диапазонов осуществляется посредством срезов аналогично тому, как это сделано в библиотеке NumPy.

Важная особенность: при использовании `loc` в выборку включаются оба граничных значения диапазона; в то же время при использовании `iloc` второе (конечное) граничное значение диапазона в выборку не включается. `iloc` не включает последний элемент, так как работает с числами и соответствует интерфейсу цикла `for`, привычному для программистов (`for(i = 0; i < n; i++)`). Что касается `loc`, то это поле работает с именами строк и столбцов. Множества имён строк и столбцов конечны. Если требуется вывести последний столбец таблицы из N столбцов, то в случае невключения граничного значения пришлось бы передавать имя N+1 столбца. Но такого столбца в таблице ещё нет, и поэтому его имя не определено. Пример использования полей `loc` и `iloc`:

```
sdf = df.loc["l1":"l3", "c1":"c2"]
print(sdf)

isdf = df.iloc[0:3, 0:2]
print(isdf)
```

Так же как и в NumPy, в pandas можно задавать диапазоны с определённым шагом (`df.iloc[0:4:2, 0:2]`) или опускать граничные значения, что будет соответствовать началу или концу исходной таблицы (`df.iloc[:3, 1:]`).

Можно применять оператор `[]` напрямую к объекту таблицы. В этом случае будет осуществляться выборка столбцов по их именам. Если применить оператор `[]` к столбцу, будет осуществляться выборка его строк по их номерам.

Имена строк и столбцов нельзя изменять присвоением. Для этого используется метод `rename()`, в который передаётся словарь, ключами которого являются старые имена, а значениями — новые имена. Параметр `inplace` позволяет применить изменения к теку-

щему объекту. Если его не установить, метод вернёт новый объект с изменёнными именами, а имена старого объекта останутся прежними. В именах строк и столбцов можно использовать форматирование LaTeX. Но отображаться в удобном виде оно будет только при работе через IPython (Jupyter).

6.5 Операции с таблицами

Некоторые методы pandas предполагают, что колонки определяют тип данных, а в строках ниже содержатся однотипные данные в одинаковом формате. Например, первый столбец содержит метки времени, а в последующих столбцах записаны результаты измерений для соответствующих моментов времени. Исходные данные иногда форматированы иначе (перечислены в строку). В таких случаях можно воспользоваться методом `transpose()`, который позволяет транспонировать таблицу. Пример транспонирования таблиц:

```
print("Таблица до транспонирования:\n", df)
tdf = df.transpose()
print("Таблица после транспонирования:\n", tdf)
```

Для добавления нового столбца достаточно сделать присвоение массива по новому индексу. Этот массив может являться результатом преобразования набора уже существующих столбцов:

```
df["<t>"] = (df.iloc[:,0] + df.iloc[:,1] + df.iloc[:,2]) / 3
```

Для добавления новых строк, можно воспользоваться полем `at`:

```
df.at["<l1>"] = (df.loc["l1", :] +
                 df.loc["l2", :] +
                 df.loc["l3", :] +
                 df.loc["l4", :]) / 4
```

В `pandas.DataFrame` встроены методы для выполнения операций над фрагментами таблиц (вычисление средних, стандартных отклонений, минимумов, максимумов...). При вызове этих методов часто нужно указывать направление (`axis`), в котором будет производиться соответствующая операция. `axis=0` соответствует строкам, `axis=1` соответствует столбцам.

Следующий пример демонстрирует добавление столбца со значениями стандартного отклонения чисел в трёх других столбцах:

```
df["St"] = df.loc[:,"c1":"c3"].std(axis=1)
```

6.6 Пример обработки экспериментальных данных средствами Python

Пусть имеется таблица с результатами измерений некоторой зависимости $t(h)$. Для каждого значения h было проведено 3 измерения величины t. Результаты записаны в файл `results.csv`:

```
h        t1        t2        t3
10.0        3.94        3.77        4.11
17.5        2.81        2.95        3.10
24.0        2.35        2.48        2.62
28.0        2.36        2.19        2.22
30.0        2.25        2.24        2.11
```

Имеется модель, согласно которой величина $a = \frac{1}{t^2}$ линейно зависит от величины h: $a = kh + b$. Требуется на основе измеренных данных получить оценки для значений параметров модели k и h.

Основные этапы работы:

1. Вычисление среднего значения $\langle t \rangle$ для каждого значения h.
2. Вычисление погрешности $S_{<t>}$. Погрешность вычисляется по формуле: $s_{<t>} = \sqrt{\frac{1}{N}\frac{\sum_{i=1}^{N}(<t>-t_i)^2}{N-1}} = \sqrt{\frac{1}{N}}\sigma_t$, где σ_t — стандартное отклонение величины t.
3. Вычисление значений $a = \frac{1}{t^2}$ и погрешности $S_a = \frac{2S_t}{t^3}$.
4. Определение коэффициентов модели с помощью метода наименьших квадратов с весами.
5. Построение графика.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt
import pandas as pd

# 0. Загрузка данных
data = pd.read_csv("results.csv", sep="\t")

# 1. Вычисление среднего значения <t>.
data["<t>"] = data.loc[:,"t1":"t3"].mean(axis=1)
data.loc[:,"<t>"] = data.loc[:,"<t>"].round(2)

# 2. Вычисление ошибки среднего значения S<t>.
n = data.loc[:,"t1":"t3"].shape[1]
data["S<t>"] = data.loc[:,"t1":"t3"].std(axis=1)
data["S<t>"] = data["S<t>"] * np.sqrt(1/n)
data.loc[:,"S<t>"] = data.loc[:,"S<t>"].round(2)
```

```
# 3. Вычисление a = 1/t^2 и погрешности Sa
data["a"] = (1/(data["<t>"]**2)).round(3)
data["Sa"] = (2*data["S<t>"]/(data["<t>"]**3)).round(3)
print(data)

# 4. Определение коэффициентов модели с помощью МНК.
# a = k * h + b; h - аргумент, k, b... коэффициенты модели
def model_func(h, k, b):
    return k*h + b

coeffs, cov = opt.curve_fit(model_func,
                            data["h"], data["a"],
                            sigma=data["Sa"],
                            absolute_sigma=True)
coeff_errors = np.sqrt(np.diag(cov))

print("\nРезультаты аппроксимации:")
coeffs[0] = np.round(coeffs[0], 4)
coeff_errors[0] = np.round(coeff_errors[0], 4)
coeffs[1] = np.round(coeffs[1], 3)
coeff_errors[1] = np.round(coeff_errors[1], 3)

# Порядок коэффициентов в массиве соответствует
# порядку коэффициентов в model_func().
print("k  = ", coeffs[0])
print("Sk = ", coeff_errors[0])
print("b  = ", coeffs[1])
print("Sb = ", coeff_errors[1])

# 5. Построение графика
s = ("$a = kh + b$\n" +
"$k = (" + str(round(coeffs[0] * 1000, 1)) +
           " \pm " +
            str(round(coeff_errors[0] * 1000, 1)) +
            ")*10^{-3}$\n" +
"$b = (" + str(int(coeffs[1] * 1000)) +
            " \pm " +
            str(int(coeff_errors[1] * 1000)) +
            ")*10^{-3}$")

# Аппроксимированная зависимость.
# Поскольку это прямая, достаточно двух точек.
h_fit = np.array([8, 32])
a_fit = h_fit * coeffs[0] + coeffs[1]
```

```
# Экспериментальные данные
plt.errorbar(data["h"], data["a"], data["Sa"],
             marker="o", linestyle="None",
             color="black", capsize=5)
plt.plot(h_fit, a_fit, color="black")
plt.text(10, 0.175, s, bbox=dict(boxstyle="square",
                                 ec=(0., 0., 0.),
                                 fc=(1., 1., 1.)))
plt.title("Зависимость величины $a$ от высоты")
plt.xlabel("Высота $h$, см.")
plt.ylabel("a, $c^{-2}$")
plt.ylim(0, 0.25)
plt.grid()

plt.savefig("plot.pdf")
plt.show()
```

Результат выполнения программы:

```
      h    t1    t2    t3   <t>  S<t>      a     Sa
0  10.0  3.94  3.77  4.11  3.94  0.10  0.064  0.003
1  17.5  2.81  2.95  3.10  2.95  0.08  0.115  0.006
2  24.0  2.35  2.48  2.62  2.48  0.08  0.163  0.010
3  28.0  2.36  2.19  2.22  2.26  0.05  0.196  0.009
4  30.0  2.25  2.24  2.11  2.20  0.05  0.207  0.009

Результаты аппроксимации:
k  =  0.0072
Sk =  0.0003
b  =  -0.008
Sb =  0.006
```

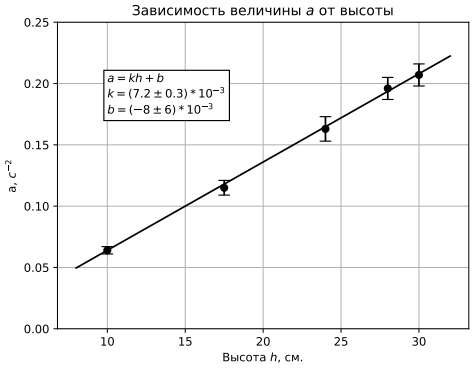


Рис. 11: График, полученный в результате выполнения примера обработки экспериментальных данных

7 Свёртка дискретных сигналов

Операция свёртки часто используется в различных задачах цифровой обработки сигналов. Свёртка применяется к паре функций (сигналов). Результатом свёртки также является функция. Формальная запись определения операции свёртки $(f * g)$ функций f и g имеет вид:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{+\infty} f(t-\tau)g(\tau)d\tau$$

Некоторые свойства свёртки:

- *Коммутативность*: $f * g = g * f$
- *Ассоциативность*: $f * (g * h) = (f * g) * h)$
- *Линейность*:
  - $f * (g_1 + g_2) = f * g_1 + f * g_2$
  - $f * (\alpha g) = \alpha(f * g)$, α — число

7.1 Свёртка сигналов, определённых на конечном отрезке времени

Реальные сигналы (назовём их f^r и g^r), которые могут быть получены в результате измерений, определены на конечных отрезках времени $[t_{f_{min}}, t_{f_{max}}]$ и $[t_{g_{min}}, t_{g_{max}}]$. Поэтому формально применить свёртку к таким сигналам нельзя. Вместо них нужно использовать дополненные сигналы f и g, которые определены для любых значений аргумента t, совпадают с функциями f^r и g^r на отрезках $[t_{f_{min}}, t_{f_{max}}]$ и $[t_{g_{min}}, t_{g_{max}}]$ соответственно и доопределены во всех остальных точках таким образом, чтобы интеграл свёртки сходился. Чаще всего используется дополнение нулями.

В случае дополнения сигналов нулями формула для вычисления свёртки принимает вид:

$$(f * g)(t) = \int_{t_{f_{min}}}^{t_{f_{max}}} f(\tau)g(t-\tau)d\tau = \int_{t_{g_{min}}}^{t_{g_{max}}} f(t-\tau)g(\tau)d\tau$$

При этом функция, полученная в результате свёртки, будет отлична от нуля на отрезке $[t_{f_{min}} + t_{g_{min}}, t_{f_{max}} + t_{g_{max}}]$. Область $T_{full} = [t_{f_{min}} + t_{g_{min}}, t_{f_{max}} + t_{g_{max}}]$ иногда называют *полной (full)* областью

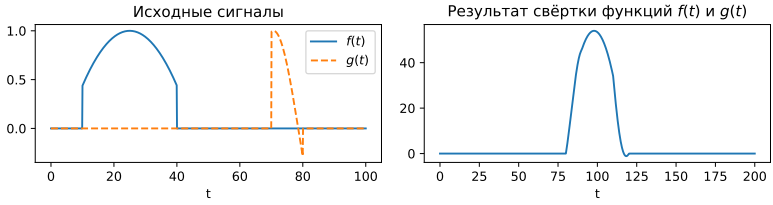


Рис. 12: Результат свёртки сигналов $f(t)$ и $g(t)$

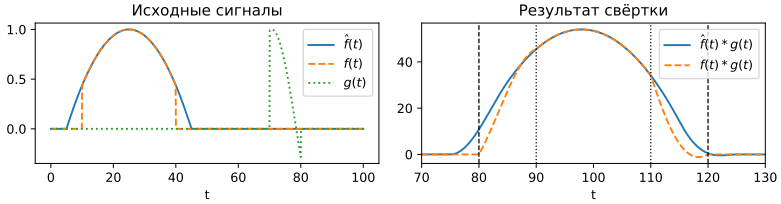


Рис. 13: Изменение свёртки в случае сужения области определения одного из исходных сигналов (сигнал $\hat{f}(t)$ является фрагментом сигнала $f(t)$)

свёртки. Пример исходных сигналов, доопределённых нулями, и результата их свёртки изображён на рисунке 12.

При вычислении свёртки вблизи граничных точек $t_{f_{min}} + t_{g_{min}}$ и $t_{f_{max}} + t_{g_{max}}$ используются те области сигналов f и g, которых не было в реальных сигналах f^r и g^r, и которые поэтому были доопределены нулями. Таким образом, для вычисления точек вблизи границы используются не только реальные сигналы, но и их «додуманные» продолжения.

Пусть область определения сигнала f^r шире области определения сигнала g^r, и кроме того, сигнал f^r является фрагментом сигнала $\hat{f}^r$. То есть имеется сигнал $\hat{f}^r$, он определён на отрезке $[t_{\hat{f}_{min}}, t_{\hat{f}_{max}}]$, совпадает с f^r на отрезке $[t_{f_{min}}, t_{f_{max}}]$, и при этом выполняются условия $t_{\hat{f}_{min}} < t_{f_{min}} < t_{g_{min}}$ и $t_{g_{max}} < t_{f_{max}} < t_{\hat{f}_{max}}$. Тогда свёртки $f * g$ и $\hat{f} * g$ будут отличаться вблизи граничных точек $t_{f_{min}} + t_{g_{min}}$ и $t_{f_{max}} + t_{g_{max}}$. В то же время в центральной области $f * g$ и $\hat{f} * g$ будут совпадать, поскольку для вычисления точек в этой области используются только точки реальных сигналов f^r и g^r, а точки из дополненных нулями областей не используются. Изменение свёртки при сужении области определения одного из сигналов показано на рисунке 13.

Область, в которой свёртка вычисляется только с использовани-

ем точек, которые входят в реальные сигналы f^r и g^r, называют *валидной (valid)* областью свёртки. Эта область соответствует отрезку $[min(t_{f_{min}} + t_{g_{max}}, t_{f_{max}} + t_{g_{min}}), max(t_{f_{min}} + t_{g_{max}}, t_{f_{max}} + t_{g_{min}})]$.

Стоит отметить, что размер валидной области растёт с разностью длин сигналов f^r и g^r. Если сигналы f^r и g^r имеют одинаковую длину, валидная область состоит всего из одной точки. Если же один сигнал значительно длиннее другого, то размер валидной области будет сопоставим с размером более длинного сигнала.

7.2 Свёртка дискретных сигналов

Реальные цифровые сигналы не только определены на конечных отрезках времени, но также и дискретны. То есть функции f^r и g^r определены на конечных множествах равномерно распределённых точек $\{t_{s_f}, t_{s_f+1}, ..., t_{s_f+N_f-1}\}$ и $\{t_{s_g}, t_{s_g+1}, ..., t_{s_g+N_g-1}\}$ соответственно (s_i — индекс первого временного отсчёта сигнала; N_i - количество временных отсчётов сигнала; $t_{s_i+k+1} - t_{s_i+k} = h_t$, где h_t — фиксированный шаг по времени при любом k). Поэтому в формуле для вычисления свёртки интеграл заменяется суммой:

$$(f * g)_n = \sum_{m=s_f}^{s_f+N_f-1} f_m g_{n-m} = \sum_{m=s_g}^{s_g+N_g-1} f_{n-m} g_m$$

Полная область свёртки начнётся в точке, которая имеет индекс $s_{f*g_{full}} = s_f + s_g + 1$. Количество точек в этой области будет $N_{f*g_{full}} = N_f + N_g - 1$. Они будут соответствовать набору временных отсчётов $\{t_{s_{f*g_{full}}}, t_{s_{f*g_{full}}+1}, ..., t_{s_{f*g_{full}}+N_{f*g_{full}}-1}\}$.

Валидная область свёртки, в свою очередь, начнётся в точке с индексом $s_{f*g_{valid}} = s_f + s_g + min(N_f, N_g)$ и будет содержать $N_{f*g_{valid}} = |N_f - N_g| + 1$ точку, что будет соответствовать набору временных отсчётов $\{t_{s_{f*g_{valid}}}, t_{s_{f*g_{valid}}+1}, ..., t_{s_{f*g_{valid}}+N_{f*g_{valid}}-1}\}$.

Пример свёртки двух дискретных сигналов с указанием границ областей изображён на рисунке 14.

7.3 Свёртка дискретных сигналов в Python средствами NumPy

Для свёртки сигналов в пакете NumPy используется функция `convolve()` [7]. В качестве первых двух аргументов эта функция

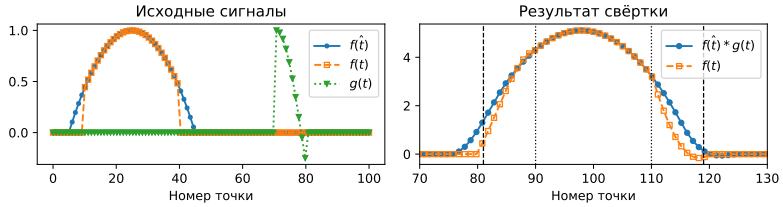


Рис. 14: Пример свёртки двух дискретных сигналов. Вертикальные линии указывают границы областей

принимает два массива с сигналами, которые требуется свернуть. По умолчанию `convolve()` возвращает полную область свёртки.

Изменить возвращаемую область свёртки можно, задав значение параметра `mode`:

- `"full"` — полная область свёртки.
- `"same"` — область, размер которой равен размеру максимального из входных массивов.
- `"valid"` — валидная область свёртки.

Что касается режима `same`, то при его использовании возвращается $N_{f*g_{same}} = max(N_f, N_g)$ точек, а смещение первой точки равно $s_{f*g_{same}} = \frac{min(N_f, N_g)+1}{2}$.

Рассмотрим пример свёртки сигналов на языке Python. Для начала нужно сгенерировать два сигнала:

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

t_f = np.linspace(0, 10, 100) # Набор значений аргумента
# f - фрагмент гармонического сигнала
s_f = 0          # Смещение - 0
N_f = len(t_f)   # Длина - 100 точек
f = np.cos(t_f)

# g - прямоугольное окно
s_g = 0          # Смещение - 0
N_g = 10         # Длина - 10 точек
g = np.ones(N_g)
```

Полученные сигналы изображены на рисунке 15.

Произведём свёртку в трёх различных режимах:

```
conv_full = np.convolve(f, g, mode="full")
conv_same = np.convolve(f, g, mode="same")
conv_valid = np.convolve(f, g, mode="valid")
```

Если вывести содержимое массивов, полученных в результате свёртки, на график, они не будут совпадать друг с другом (см. рисунок 16 А).

Рис. 15: Исходные сигналы, используемые в примере

Чтобы убедиться в том, что результаты свёртки во всех трёх режимах совпадают друг с другом в тех точках, где они определены, нужно вычислить смещения и совместить полученные сигналы с учётом полученных значений (результат изображён на рисунке 16 Б):

```
# Вычисляем смещения
s_full = s_f + s_g + 1
s_same = (min(N_f, N_g) + 1) // 2
s_valid = s_f + s_g + min(N_f, N_g)

# Вычисляем длины сигналов
N_full = N_f + N_g - 1
N_same = max(N_f, N_g)
N_valid = np.abs(N_f - N_g) + 1

# Области определения
t_full = range(s_full, s_full + N_full)
t_same = range(s_same, s_same + N_same)
t_valid = range(s_valid, s_valid + N_valid)

# Построение графика с учётом смещений
plt.plot(t_full, conv_full, label="full")
plt.plot(t_same, conv_same, label="same")
plt.plot(t_valid, conv_valid, label="valid")
plt.legend()
plt.show()
```

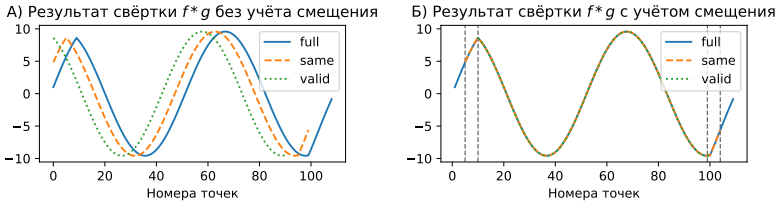


Рис. 16: Результаты свёртки сигналов в режимах full, same и valid без учёта смещения (А) и с учётом смещения (Б)

Теперь в валидной области все три сигнала совпадают. Отличия

возникают только за пределами валидной области и обусловлены краевыми эффектами. Поведение сигнала в областях краевых эффектов не является «естественным продолжением» поведения сигнала в валидной области и обусловлено тем, что в этих областях свёртка вычислялась по «додуманным» данным.

8 Основы корреляционного анализа

Корреляционная функция — это функция, которая позволяет оценить степень совпадения двух сигналов. Корреляционная функция дискретного сигнала вычисляется по следующей формуле:

$$C(f,g)_k = \sum_{n=-\infty}^{+\infty} f_{n+k} g_n^*$$

Эта формула похожа на формулу для вычисления свёртки, но отличается от неё знаками индексов временных отсчётов и комплексным сопряжением (обозначено символом *). При вычислении корреляционной функции конечных сигналов, как и в случае со свёрткой, для формального соответствия определению конечные сигналы дополняются нулями. Результат вычисления корреляционной функции точно так же содержит полную (*full*) область, соответствующую ненулевыми значениям корреляционной функции, и валидную (*valid*) область, при расчёте которой использовались только точки, присутствовавшие во входных сигналах, и не использовались доопределённые нулями точки.

Для вычисления корреляционных функций можно использовать метод `correlate()` из пакета NumPy. Этот метод, в отличие от метода `convolve()`, по умолчанию использует режим valid. Если исходные сигналы f и g содержат N_f и N_g точек соответственно, то в результате вычисления корреляционной функции $C(f,g)$ будет получена $N_{C_{fg}} = |N_f - N_g| + 1$ точка.

При использовании корреляционной функции для поиска фрагментов сигнала g, входящих в более длинный сигнал f, важно учитывать смещение $s_{C_{fg}} = s_f + (N_g - 1)/2$. Массив, полученный в результате вычисления корреляционной функции, нужно сдвинуть на величину $s_{C_{fg}}$, чтобы координаты максимумов корреляционной функции совпадали со смещениями сигнала g относительно сигнала f, при которых совпадение между f и g в соответствующих областях будет максимальным.

8.1 Применение корреляционной функции для поиска сигнала на фоне шума

В качестве примера использования корреляционной функции рассмотрим задачу поиска сигнала на фоне шума по известному шаблону. Пусть имеется сигнал $f_0(t)$, который содержит несколько импульсов $f_p(t)$. Форма импульсов $f_p(t)$ известна. Но данные о том, как они расположены во времени, отсутствуют. Измеряется сигнал $f(t)$, который представляет собой сумму сигнала $f_0(t)$ и шума $\xi(t)$ с нормальным распределением и нулевым средним. Форма импульсов $f_p(t)$ и зашумлённый сигнал $f(t)$ известны. Требуется определить положения центров импульсов в исходном сигнале $f_0(t)$.

Поскольку шум имеет нормальное распределение и нулевое среднее, при вычислении корреляционной функции $C(f, f_p)$ вклад шумовой составляющей будет невелик. Получившаяся функция будет близка к функции $C(f_0, f_p)$, которая могла бы быть получена в отсутствие шума. Поэтому положения импульсов можно найти по пикам корреляционной функции. Пример кода на языке Python:

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

# Создание исходных сигналов
t = np.linspace(0, 20, 1000)
f_0 = (0.5 + 0.5 * np.sin(t)) ** 20
f_p = np.copy(f_0[50:111])
xi = np.random.normal(0, 1, len(f_0))
f = f_0 + xi

# Вычисляем корреляционную функцию
c = np.correlate(f, f_p)
s_c = (len(f_p) - 1) // 2 # Смещение

# Визуализация
plt.figure(figsize=(5, 5))
plt.subplot(2, 1, 1)
plt.plot(t[50:111] + 4*np.pi, f_p-4, label="Шаблон $f_p$")
plt.plot(t, f, label="Сигнал с шумом $f$", linestyle=":")
plt.plot(t, f_0, label="Сигнал $f_0$", linestyle = "-")
plt.legend()
plt.xlim(0, 20)
```

```
plt.subplot(2, 1, 2)
plt.plot(t[s_c:-s_c], c, color="red", label="$C(f, f_p)$")
plt.xlabel("Время, с")
plt.legend()
plt.xlim(0, 20)

plt.show()
```

Сигналы и корреляционная функция, полученные в результате выполнения кода из примера, изображены на рисунке 17. Максимумы корреляционной функции $C(f, f_p)$ хорошо соответствуют положениями пиков в исходном сигнале f_0, несмотря на достаточно сильный шум сигнала f.

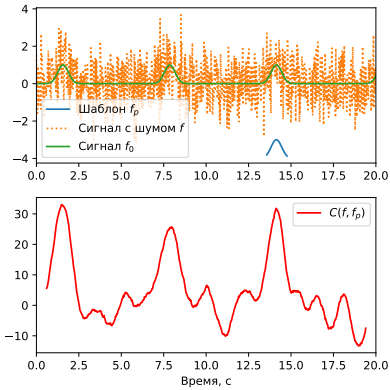


Рис. 17: Поиск сигнала на фоне шума корреляционным методом. На верхнем графике изображён исходный сигнал f_0, сигнал f после добавления шума и шаблон f_p для поиска. На нижнем графике представлена корреляционная функция $C(f, f_p)$

9 Базовые методы цифровой фильтрации сигналов

Одной из основных задач фильтрации сигналов является устранение шумов. Простыми, но в то же время достаточно эффективных методами цифровой фильтрации сигналов являются:

- *Скользящее среднее*
- *Экспоненциальное сглаживание*
- *Медианная фильтрация*

9.1 Скользящее среднее

Фильтрация сигналов посредством *скользящего среднего* осуществляется путём вычисления значения выходного сигнала $\bar{f}(t_k)$ в точке t_k, как среднего значения исходного сигнала f внутри окна $W = [t_{k-(d-1)/2}, t_{k+(d-1)/2}]$:

$$\bar{f}(t_k) = \frac{1}{d} \sum_{m=k-(d-1)/2}^{k+(d-1)/2} f(t_m)$$

Такое усреднение можно представить в виде свёртки $\bar{f} = f * k$, где k — ядро свёртки, состоящее из d точек со значением $\frac{1}{d}$ (константа). Пример фильтрации сигнала методом скользящего среднего:

```
import numpy as np

def running_average(signal, k_dim):
    # k_dim - размер скользящего окна
    k = np.ones(k_dim) / k_dim
    return np.convolve(signal, k, mode="same")

# Формирование исходного сигнала
t = np.linspace(0, 10, 100)
freq = 0.2 # Гц
f_0 = np.sin(2 * np.pi * freq * t)
xi = np.random.normal(0, 0.5, len(f_0))
f = f_0 + xi # Смесь сигнала с шумом

# Фильтрация
f_a = running_average(f, 11)
```

Сигналы, полученные в результате выполнения данного примера, изображены на рисунке 18.

С увеличением размера окна d шум сглаживается сильнее, но при этом вместе с шумом начинает сглаживаться и полезный сигнал. Величину d следует выбирать таким образом, чтобы она превышала характерное время изменения шума, но при этом была значительно меньше характерного времени изменения полезного сигнала. Зависимость формы сглаженного сигнала от размера скользящего окна изображена на рисунке 19.

9.2 Экспоненциальное сглаживание

Экспоненциальное сглаживание подразумевает свёртку исходного сигнала с весовой функцией, веса которой убывают по экспо-

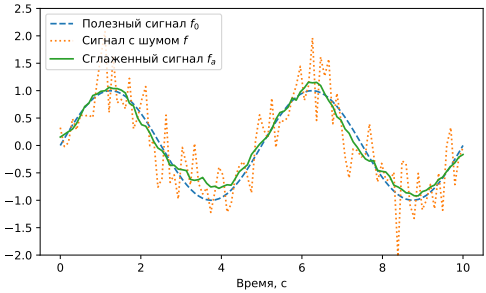


Рис. 18: Пример фильтрации сигнала методом скользящего среднего

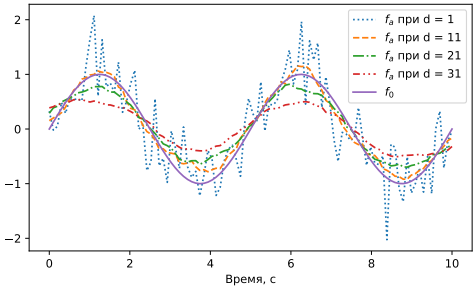


Рис. 19: Зависимость формы сглаженного сигнала f_a от размера скользящего окна d; f_0 — незашумлённый полезный сигнал

ненциальному закону:

$$\bar{f}(t_k) = (1-q)\sum_{m=0}^{\infty} q^m f(t_{k-m}),\ q \in (0,1)$$

Экспоненциальный фильтр учитывает недавно полученные точки с большим весом, чем точки, которые были получены ранее.

Особенностью экспоненциального фильтра является то, что его можно преобразовать к удобному для вычислений виду:

$$\bar{f}(t_k) = (1-q)\sum_{m=0}^{\infty} q^m f(t_{k-m}) =$$

$$= (1-q)f(t_m) + (1-q)\sum_{m=1}^{\infty} q^m f(t_{k-m}) =$$

$$= (1-q)f(t_k) + q\left((1-q)\sum_{m=1}^{\infty} q^{m-1} f(t_{k-1-(m+1)})\right) =$$

$$= (1-q)f(t_k) + q\left((1-q)\sum_{n=0}^{\infty} q^{n} f(t_{k-1-n})\right) =$$

$$= (1-q)f(t_k) + q\bar{f}(t_{k-1})$$

Таким образом, зная значение усреднённого сигнала $\bar{f}(t_{k-1})$ на предыдущем шаге по времени, можно за несколько операций вычислить значение усреднённого сигнала на текущем шаге. Функция, реализующая экспоненциальный фильтр на Python, может выглядеть следующим образом:

```
def exp_average(signal, q):
    # q - коэффициент сглаживания
    # должен принимать значения из (0, 1)
    f_s = [signal[0]]
    for i in range(1, len(signal)):
        f_s.append((1 - q) * signal[i] + q * f_s[i - 1])
    return f_s
```

Результат применения этой функции с коэффициентом сглаживания $q = 0,8$ к сигналу из предыдущего примера изображён на рисунке 20.

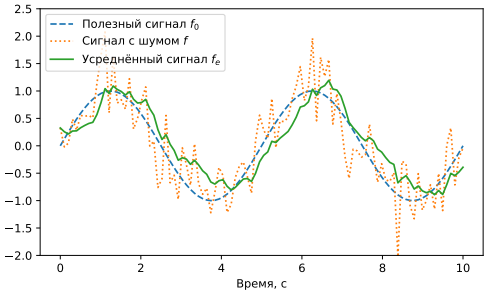


Рис. 20: Пример фильтрации методом экспоненциального сглаживания

Так же как и в случае со скользящим средним увеличение параметра сглаживания q приводит не только к усилению устранения шума, но и к сглаживанию полезного сигнала f_0. Кроме то-

го, при больших значениях q экспоненциальный фильтр может заметно смещать (задерживать) сглаженный сигнал f_e относительно исходного полезного сигнала f_0. Зависимость формы сглаженного сигнала от параметра экспоненциального сглаживания q изображена на рисунке 21.

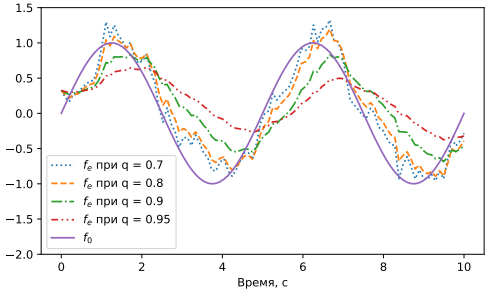


Рис. 21: Зависимость формы сглаженного сигнала f_e от значения параметра экспоненциального сглаживания q; f_0 — незашумлённый полезный сигнал

9.3 Медианная фильтрация

В случае *медианной фильтрации*, также как и в случае со скользящим средним, выходное значение фильтра $\bar{f}(t_k)$ вычисляется на основе точек, попадающих в окно $W = [t_{k-(d-1)/2}, t_{k+(d-1)/2}]$, где d — размер окна. Но в отличие от скользящего среднего точки внутри окна не усредняются, а сортируются, после чего выбирается точка, находящаяся в середине отсортированного массива:

```
[7, 4, 6, 5, 4] -> [4, 4, 5, 6, 7] -> 5
```

Лучше всего медианный фильтр подходит для устранения импульсных помех но применим и для других видов шума. Алгоритм медианной фильтрации на языке Python можно реализовать следующим образом:

```
def median_filter(signal, d):
    # d - размер окна. Должен быть нечётным.
    if d % 2 == 0:
        raise ValueError

    # Расширение исходного массива для обеспечения
    # совпадения длин входного и выходного массивов
    offset = (d - 1) // 2
    f_b = np.zeros(offset) + signal[0]
    f_e = np.zeros(offset) + signal[-1]
    ext_signal = np.concatenate((f_b, signal, f_e))

    # Фильтрация
    f_s = []
    for i in range(len(signal)):
        f_s.append(np.median(ext_signal[i:i+d]))

    return f_s
```

Результат применения данной функции при использовании окна размером $d = 11$ к тестовым сигналам из предыдущих примеров изображён на рисунке 22.

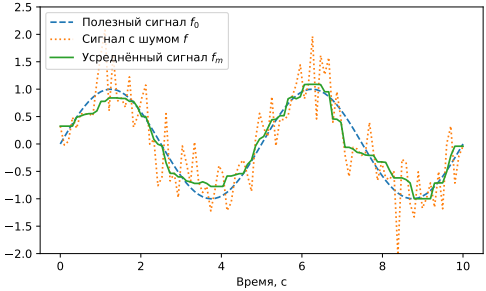


Рис. 22: Пример медианной фильтрации

Когда размер окна d оказывается сравним с характерным временем изменения полезного сигнала f_0, медианный фильтр, как и все остальные фильтры, начинает сглаживать не только шум, но и полезный сигнал. Особенностью медианного фильтра является то, что он в отличие от других фильтров не равномерно «гасит» полезный сигнал, а «срезает» максимумы и минимумы. Результат медианной фильтрации при больших значениях размера окна d, как правило, содержит большое количество отрезков постоянного уровня. Набор выходных сигналов для медианной фильтрации с разными значениями параметра d изображён на рисунке 23.

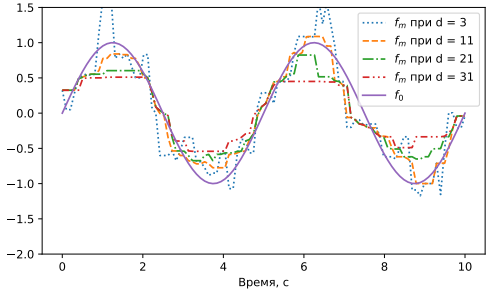


Рис. 23: Влияние размера окна d на результат медианной фильтрации.

9.4 Задание для самостоятельного выполнения

Сравнить качество фильтрации методами скользящего среднего, экспоненциального сглаживания и медианной фильтрации для различных значений параметров фильтров.

Основные этапы выполнения задания:

- Написать функции, реализующие перечисленные фильтры.
- Сгенерировать тестовый полезный сигнал.
- Добавить шум к полезному сигналу.
- Написать функцию, вычисляющую среднеквадратичное отклонение двух последовательностей друг от друга.
- Построить зависимости среднеквадратичной ошибки сигнала, обработанного фильтром, от параметра фильтра для всех трёх методов.

10 Дискретное преобразование Фурье

Для решения некоторых задач бывает удобно представлять сигналы не в виде зависимости $g(t)$ некоторой физической величины от времени, а в виде разложения по базису, состоящему из набора функций $\varphi_n(t)$ с хорошо известными свойствами. В этом случае сигнал характеризуется набором весовых коэффициентов c_n (или функцией распределения $G(f)$). Исходная временная зависимость может быть с определённой точностью восстановлена путём суммирования базисных функций с соответствующими весовыми коэффициентами (или путём интегрирования весовой функции):

$$g(t) = \sum_{n=-\infty}^{+\infty} c_n \varphi_n(t)$$

$$g(t) = \int_{-\infty}^{+\infty} G(f)\varphi(f,t)df$$

Более подробную информацию о том, в каких случаях можно использовать такие разложения и какими свойствами они обладают, можно получить в [9] (глава 10 Ряды и интеграл Фурье).

10.1 Преобразование Фурье

Одним из наиболее часто используемых наборов базисных функций является набор гармонических функций. Для получения такого разложения используется *преобразование Фурье*. В случае непрерывных функций преобразование Фурье имеет вид:

$$G(f) = \int_{-\infty}^{+\infty} g(t)e^{-2\pi i f t}dt$$

Обратное преобразование Фурье, позволяющее получить временную зависимость, имеет похожий вид:

$$g(t) = \int_{-\infty}^{+\infty} G(f)e^{2\pi i f t}df$$

Если говорить простым языком, то преобразование Фурье позволяет узнать спектральный состав сигнала — сколько синусов и косинусов и с какими частотами нужно сложить, чтобы получить интересующий нас сигнал.

Когда требуется осуществить разложение дискретной определённой на конечном промежутке времени функции $g_m = g(t_m)$, используют *дискретное преобразование Фурье* (ДПФ).

Прямое преобразование:

$$G_k = \sum_{m=0}^{N-1} g_m e^{-2\pi i \frac{mk}{n}}$$

Обратное преобразование:

$$g_m = \frac{1}{n} \sum_{k=0}^{N-1} G_k e^{2\pi i \frac{mk}{n}}$$

10.2 Некоторые свойства дискретного преобразования Фурье

Пусть исходный сигнал задан во временной области функцией $g_n = g(t_n)$, состоящей из N точек, следующих с постоянным шагом $h_t = t_{i+1} - t_i = const$ на отрезке времени $[0, t_{max} = (N-1)h_t]$. Тогда:

- *Количество точек* в спектре $G_m = G(f_m)$ совпадает с количеством точек во временном представлении и равно N.
- *Шаг по частоте* между точками в спектре обратно пропорционален времени измерения сигнала: $h_f = f_{i+1} - f_i = \frac{1}{t_{max}}$.
- *Максимальная частота* в спектре обратно пропорциональна шагу по времени: $f_{max} = (N-1)h_f = \frac{1}{h_t}$.
- *Амплитуда* A_f пика в спектре чистого гармонического сигнала $g(t) = A_t sin(2\pi f t)$ пропорциональна амплитуде сигнала во временном представлении и количеству точек сигнала: $A_f = \frac{1}{2} A_t N$.

Таким образом, чем дольше производилось измерение исходного сигнала, тем выше будет разрешение по частоте, и наоборот — чем выше было разрешение во временной области, тем шире будет диапазон частот в спектре. Кроме того, с увеличением времени измерения растут и амплитуды пиков в спектре, это связано с тем, что в случае чистого гармонического сигнала все точки во временном представлении вносят вклад в формирование единственной точки в спектре.

Рассмотрим пример использования преобразования Фурье из программы на языке Python. Для анализа будет использоваться сигнал, представляющий собой сумму двух гармонических сигналов с разными частотами и амплитудами: $g(t) = A_1 sin(2\pi f_1 t) + A_2 sin(2\pi f_2 t)$.

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

t = np.linspace(0, 10, 5000)
f_1 = 20  # Частота первого сигнала в Гц
f_2 = 27  # Частота второго сигнала в Гц
a_1 = 2   # Амплитуда первого сигнала
a_2 = 1   # Амплитуда второго сигнала

g_1 = a_1 * np.sin(2 * np.pi * f_1 * t)
g_2 = a_2 * np.sin(2 * np.pi * f_2 * t)
g = g_1 + g_2
```

Для осуществления преобразования Фурье в NumPy используется метод `fft()` (Fast Fourier transform) из одноименного подмодуля. В общем случае спектр сигнала является комплексным. Для большинства практических задач интерес представляет только модуль спектра.

```
h_t = t[1] - t[0]  # Шаг по времени исходного сигнала
f_max = 1 / h_t    # Максимальная частота в спектре
f = np.linspace(0, f_max, len(t))  # Сетка частот в Гц

g_spectrum = np.fft.fft(g)
```

Код для визуализации сигнала во временном и спектральном представлениях:

```
plt.figure(figsize=(10, 3))
plt.subplot(1, 2, 1)
plt.title("Сигналы во временном представлении")
plt.plot(t, g, label="$g(t)$")
plt.plot(t, g_1, label="$A_1 \sin(2\pi f_1 t)$",
                 linestyle="--")
plt.plot(t, g_2, label="$A_2 \sin(2\pi f_2 t)$",
                 linestyle=":")
plt.xlabel("Время, с")
plt.xlim(0, 0.5)
plt.legend(loc=1, framealpha=1)

plt.subplot(1, 2, 2)
plt.title("Спектр сигнала (модуль)")
plt.axvline(f_1, color="k", linestyle="--", label="$f_1$")
plt.axvline(f_2, color="k", linestyle="-.", label="$f_2$")
plt.axhline(a_1 * len(f) / 2, color="g", linestyle="--",
            label="$A_{f1}=\\frac{1}{2}A_{t1}N$")
plt.axhline(a_2 * len(f) / 2, color="g", linestyle="-.",
            label="$A_{f2}=\\frac{1}{2}A_{t2}N/2$")
plt.plot(f, np.abs(g_spectrum), label="$G(f)$")
plt.xlim(0, 50)
plt.ylim(0, len(t) * 1.2)
plt.xlabel("Частота, Гц")
plt.legend(loc=1, framealpha=1)
plt.show()
```

Сигналы, полученные в результате выполнения примера, изображены на рисунке 24. Положение и высота пиков на графике модуля спектра сигнала соответствуют частотам и амплитудам гармонических составляющих исходного сигнала.

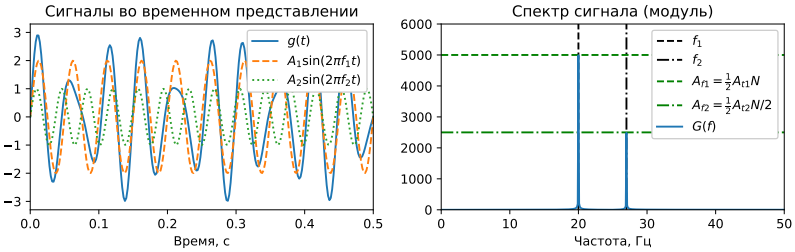


Рис. 24: Сигнал, представляющий собой сумму двух гармонических сигналов разной амплитуды, во временном и спектральном представлениях

10.3 Случай некратных частот

Если частота гармоники исходного сигнала некратна шагу сетки по частоте ($f \neq kh_f, k \in \mathbb{N}$), амплитуда соответствующего пика будет ниже ожидаемой, а сам пик будет распределён между несколькими соседними точками. Простого способа восстановить амплитуду сигнала в данном случае нет. Пример спектра сигнала, включающего в себя частоты $f_1 = 20.13$ Гц и $f_2 = 27.07$ Гц, изображён на рисунке 25.

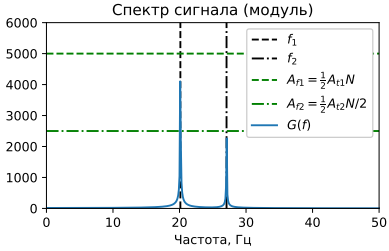


Рис. 25: Распределение спектрального пика между несколькими частотными отсчётами в случае некратных шагу сетки частот.

10.4 Перестановка отрицательных частот

Если построить график модуля спектра во всём диапазоне частот (см. рисунок 26), то можно увидеть, что вторая половина спектра ($f_{max}/2 < f \leq f_{max}$) является зеркальным отражением первой половины ($0 \leq f < f_{max}/2$). Это связано с тем, что разложение функции $g(t)$ производится не по синусам и косинусам, а по экспонентам $e^{2\pi ift}$, а экспоненты связаны с синусами и косинусами

следующим образом:

$$A\sin(2\pi ft) = A\frac{e^{2\pi ift} - e^{-2\pi ift}}{2i}, \quad A\cos(2\pi ft) = A\frac{e^{2\pi ift} + e^{-2\pi ift}}{2}$$

Этими формулами можно объяснить и наличие коэффициента $\frac{1}{2}$ в формуле для амплитуд пиков: одному синусу или косинусу соответствует не один пик с полной амплитудой $A_f = NA_t$, а два пика с половинной амплитудой $A_f = \frac{1}{2}NA_t$, соответствующие двум экспонентам.

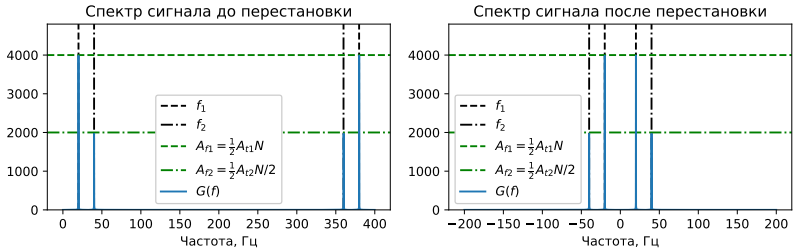


Рис. 26: Спектр сигнала до и после перестановки отрицательных частот

Область частот $f_{max}/2 < f \leq f_{max}$, можно трактовать как область, соответствующую отрицательным частотам $(-f_{max}/2 \leq f < 0)$. Бывает удобно переставить части спектра местами, чтобы получить график в диапазоне частот $-f_{max}/2 \leq f \leq f_{max}/2$. Для этого в `numpy.fft` предусмотрен метод `fftshift()`, осуществляющий перестановку. Если требуется определять частоты с большой точностью, то при данной перестановке происходит следующее преобразование частот в спектре:

$$\left[\frac{1}{2}Nh_f = \frac{f_{max} + h_f}{2}, ..., (N-1)h_f = f_{max}\right] \rightarrow \left[-\frac{f_{max} + h_f}{2}, ..., -h_f\right]$$

Пример кода для получения спектра в диапазоне частот $-f_{max}/2 \leq f \leq f_{max}/2$ и соответствующий ему набор частотных отсчётов:

```
# Формирование нового набора частотных отсчётов
h_f = f[1] - f[0]
f_shifted = np.linspace(-(f_max + h_f)/2, (f_max - h_f)/2, len(t))

# Перестановка спектра
g_spectrum_shifted = np.fft.fftshift(g_spectrum)
```

Пример спектра до и после перестановки отрицательной области частот изображён на рисунке 26.

10.5 Максимальная частота дискретного сигнала

Из описанных выше свойств дискретного преобразования Фурье следует, что полученный средствами ДПФ спектр является корректным только для сигналов, максимальная частота $f_{s_{max}}$ в спектре которых не превышает половины частоты дискретизации f_d. Это ограничение вполне естественно, так как в случае гармонического сигнала $sin(2\pi f_s t)$ при частоте дискретизации $f_d = 2f_s$, на период сигнала приходится всего две точки и с анализом сигнала проблемы возникают уже не только в спектральной области, но и во временной. Графики на рисунке 27 демонстрируют сигнал, восстановленный по измерениям, сделанным при разном отношении частоты дискретизации f_d к частоте сигнала f_s. Если $f_d/f_s > 2$, восстановленный сигнал соответствует реальному. Чем больше отношение f_d/f_s, тем точнее аппроксимация. Когда же $f_d/f_s < 2$, форма восстановленного по отсчётам сигнала имеет мало общего с формой исходного сигнала. Поэтому анализ сигналов, измеренных при отношениях $f_d/f_s < 2$, приведёт к ошибкам как во временном, так и в спектральном представлениях. Более строгое обоснование того, почему некорректно работать с сигналами, максимальная частота в спектре которых превышает половину частоты дискретизации, следует из *теоремы Котельникова*.[14]

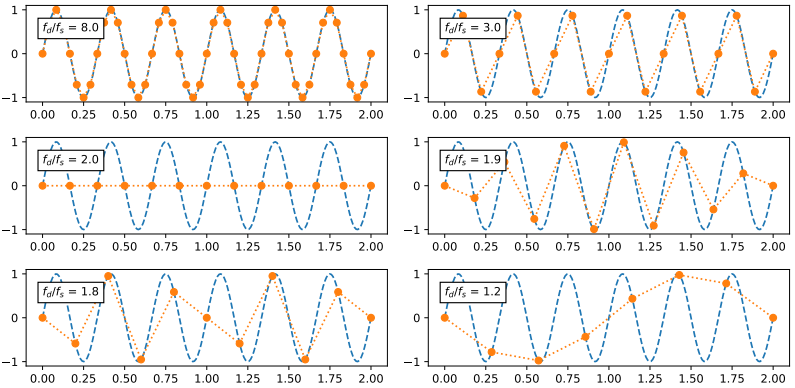


Рис. 27: Восстановление сигнала по дискретным временным измерениям при разных соотношениях частоты дискретизации f_d к характерной частоте сигнала f_s. Штрихованной линией изображён исходный сигнал. Точками изображены дискретные измерения, по которым производится восстановление

[14](https://ru.wikipedia.org/wiki/Теорема_Котельникова

10.6 Спектральная фильтрация сигналов

Часто в физических измерениях полезный сигнал находится в узкой области частот. Например, в радиосвязи сигнал обычно передаётся на фиксированной несущей частоте, а передача информации осуществляется за счёт амплитудной или фазовой модуляции, которая лишь незначительно уширяет спектр. Шумы возникают в гораздо более широком диапазоне частот, и лишь незначительная часть этого диапазона перекрывается со спектром полезного сигнала. В таких случаях значительная часть шумов может быть устранена путём спектральной фильтрации.

Идея *спектральной фильтрации* заключается в том, что из спектра исходного сигнала можно удалить те частоты, на которых полезного сигнала точно нет, и оставить лишь те, на которых полезный сигнал может быть. В этом случае во временном представлении исчезнет значительная часть шумового сигнала. Составляющая полезного сигнала при этом сильно не изменится.

Спектральную фильтрацию можно осуществлять различными методами (в том числе в процессе измерения с использованием аппаратных фильтров). Но одним из самых наглядных способов является фильтрация с использованием преобразования Фурье.

Рассмотрим спектральную фильтрацию с использованием преобразования Фурье на конкретном примере. Пусть имеется полезный сигнал $g(t)$ вида

$$g(t) = A_1 sin(2\pi f_1 t) + A_2 sin(2\pi f_2 t)$$

Регистрируется сигнал $g_n(t)$, который представляет собой сумму полезного сигнала $g(t)$ и шума $\xi(t)$, спектр которого лишь частично перекрывается с частотами полезного сигнала. Будет использоваться шум с нормальным распределением и нулевым средним. Пример такого сигнала во временном и спектральном представлениях изображён на рисунке 28. Задача заключается в том, чтобы получить оценку полезного сигнала $g(t)$, имея только измеренный сигнал $g_n(t)$. Код на языке Python, позволяющий сгенерировать описанные выше сигналы и получить их спектры, приведён ниже.

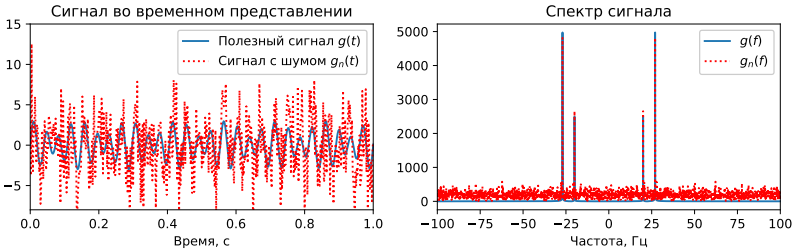


Рис. 28: Полезный сигнал $g(t)$ и измеренный сигнал $g_n(t)$ во временном и спектральном представлениях

```
import numpy as np

t = np.linspace(0, 10, 5000)  # Временные отсчёты в с
f_1 = 20; f_2 = 27            # Частоты в Гц
a_1 = 1.; a_2 = 2.            # Амплитуды

g_1 = a_1 * np.sin(2*np.pi*f_1*t)
g_2 = a_2 * np.sin(2*np.pi*f_2*t)
xi = np.random.normal(0, 3, len(t))  # Шум
g_n = g_1 + g_2 + xi

# Набор частотных отсчётов
h_t = t[1] - t[0]; h_f = 1/t[-1]
f_max = 1/h_t
f = np.linspace(-(f_max + h_f)/2, (f_max - h_f)/2, len(t))

# Преобразование Фурье с перестановкой отрицательных частот
g_n_spectrum = np.fft.fft(g_n)
g_n_spectrum = np.fft.fftshift(g_n_spectrum)
```

Будем считать, что полезный сигнал находится в диапазоне частот $[-28; -18] \cup [18; 28]$ Гц. Поскольку в данном примере сигнал содержит всего две частоты, можно было бы взять более узкий диапазон, покрывающий только эти две частоты. Но реальные сигналы, как правило, устроены сложнее и имеют более сложный спектр. Поэтому диапазон для фильтрации имеет смысл выбирать с небольшим запасом.

Чтобы снизить уровень шума, все частоты в спектре за пределами этого диапазона нужно удалить (сделать амплитуду равной нулю). Таким образом будет получен спектр $g_f(f)$. Затем с помощью обратного преобразования Фурье нужно получить временное представление $g_f(t)$. Это можно сделать следующим образом:

```
# Маска, определяющая набор частот, которые не будут удалены
f_mask = np.logical_or(np.logical_and(f > -28, f < -18),
                       np.logical_and(f >  18, f <  28))

# Удаление всех частот, кроме тех, что закрыты маской
g_f_spectrum = np.where(f_mask, g_n_spectrum, 0)

# Обратное преобразование Фурье
g_f_spectrum_s = np.fft.fftshift(g_f_spectrum)
g_f = np.fft.ifft(g_f_spectrum_s)
```

Результат спектральной фильтрации изображён на рисунке 29. Поскольку часть шума попала в диапазон частот полезного сигнала, сигнал $g_f(t)$ отличается от исходного полезного сигнала $g(t)$. Тем не менее сигнал $g_f(t)$ гораздо ближе к полезному сигналу $g(t)$, чем необработанный результат измерения $g_n(t)$.

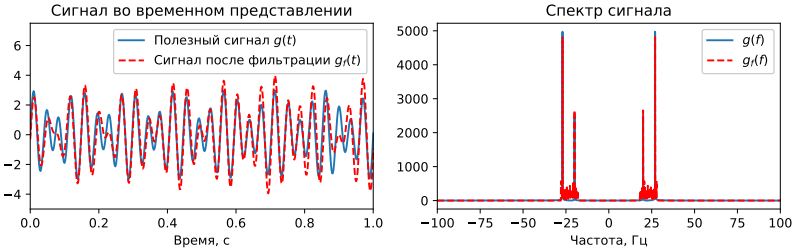


Рис. 29: Полезный сигнал $g(t)$ и сигнал $g_f(t)$, полученный в результате спектральной фильтрации, во временном и спектральном представлениях

10.7 Задание для самостоятельного выполнения

Ниже приведён код, который генерирует два wav-файла, в которых записан узкополосный звуковой сигнал. В одном из файлов на полезный сигнал наложен широкополосный шум. При желании файлы можно воспроизвести с помощью стандартного музыкального проигрывателя.

Требуется определить полосы частот, в которых содержится полезный сигнал и осуществить подавление шума в тех областях спектра, где полезного сигнала нет. Сигнал с подавленными шумами нужно сохранить в виде wav-файла. При воспроизведении полученного файла не должно быть слышно шипения.

Код, генерирующий входные звуковые файлы:

```
import numpy as np
from scipy.io import wavfile

f_d = 22050  # Частота дискретизации, точек/с
t_max = 4    # с
t = np.linspace(0, t_max, int(t_max * f_d))
fs = np.random.randint(200, 800, size=3)
fs[2] = fs[1] + np.random.randint(1, 3)
a = 20

g = a * (np.sin(2 * np.pi * fs[0] * t) +
         np.sin(2 * np.pi * fs[1] * t) +
         np.sin(2 * np.pi * fs[2] * t))
g_n = g + np.random.normal(0, 25, len(g))

wavfile.write("g.wav", f_d, np.array(g, dtype="int8"))
wavfile.write("g_n.wav", f_d, np.array(g_n, dtype="int8"))
```

Заготовка решения:

```
import numpy as np
from scipy.io import wavfile

f_d, g_n = wavfile.read("g_n.wav")
print("Discretization frequency f_d = {} Hz".format(f_d))

# Тут должна быть фильтрация
g_f = g_n

wavfile.write("g_f.wav", f_d, np.array(g_f, dtype="int8"))
```

11 Использование графиков Matplotlib в приложениях с графическим интерфейсом Qt

Библиотека Qt является одной из наиболее популярных библиотек для создания графических интерфейсов. Достаточно часто возникает необходимость встраивания графиков, построенных средствами Matplotlib, в приложения с интерфейсом, разработанным с использованием Qt. В данном разделе мы рассмотрим, как это можно сделать.

11.1 Создание приложения Qt

Перед тем, как добавлять графики, нужно создать заготовку приложения с графическим интерфейсом Qt. Ниже приведён код, необходимый для создания простейшего приложения, состоящего всего из одного пустого окна.

```
import sys
import PyQt5.QtWidgets as qt

app = qt.QApplication(sys.argv)
main_window = qt.QWidget()
main_window.setWindowTitle("Простое приложение Qt")
main_window.resize(600, 400)

### Место для добавления дополнительного кода ###

main_window.show()
sys.exit(app.exec_())
```

11.2 Встраивание графиков Matplotlib в интерфейс Qt

Для добавления графика Matplotlib в созданное ранее окно Qt нужно:

1. Импортировать Matplotlib и специальный Qt-виджет холста (canvas) для графиков Matplotlib. Также для подготовки данных потребуется NumPy.

   ```
   import matplotlib.backends.backend_qt5agg as mbq
   import matplotlib.pyplot as plt
   import numpy as np
   ```

2. Создать холст (figure) Matplotlib, создать на этом холсте область (или несколько областей) для построения (subplot), настроить область построения.

   ```
   fig = plt.figure(figsize=(4, 3), dpi=72)
   ax = fig.add_subplot(1, 1, 1)
   ax.set_title("Тестовый сигнал")
   ax.set_xlabel("Время, с")
   ax.set_ylabel("Напряжение, В")
   ```

   Созданный холст является объектом Matplotlib и пока никак не связан с созданным выше окном Qt.

3. Создать Qt-виджет холста (FigureCanvas) для встраивания

графика Matplotlib в окно Qt. Созданный ранее холст (`fig`) нужно передать в качестве аргумента в инициализатор виджета. Созданный виджет нужно связать с окном, на котором он должен отображаться. При необходимости можно задать размер и положение холста в окне.

```
fc = mbq.FigureCanvasQTAgg(fig)
fc.setParent(main_window)
fc.resize(400, 300)
fc.move(20, 20)
```

Структура созданных объектов изображена на рисунке 30.

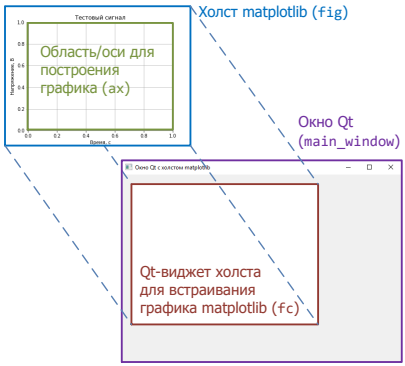


Рис. 30: Структура графических объектов при встраивании графика Matplolib в окно Qt

11.3 Добавление данных на график

Объект области построения графиков Matplotlib (в примере выше были созданы оси с именем `ax`) имеет почти такой же набор методов, как и объект `matplotlib.pyplot`. В частности, для построения обычного графика используется метод `plot()`. Пример построения графика гармонического сигнала:

```
freq = 0.5; phase = np.pi / 2
t = np.linspace(0, 10, 1000)
signal = np.sin(2*np.pi*freq*t + phase)
ax.plot(t, signal, label="Сигнал", color="green")
ax.legend(loc=1)
```

Если на графике уже имеются данные, и эти данные устарели (производится перестроение графика), то их можно удалить. Для

этого нужно получить список всех кривых, добавленных в область построения (поле `lines`) и последовательно вызвать для каждой из кривых метод `remove()`:

```
for line in ax.lines:
    line.remove()
```

Полностью очистить график можно, вызвав метод `ax.clear()`. В этом случае очистятся не только данные, но и настройки области построения (подписи осей, линии сетки, масштаб...).

Библиотека Matplotlib устроена таким образом, что она сначала рисует оси, а после этого добавляет все остальные объекты. В результате некоторые элементы графика могут оказаться за пределами холста. Чтобы этого не произошло, после построения всех кривых нужно вызывать метод `plt.tight_layout()`. Matplotlib впишет график в холст и добавит отступы от краёв холста.

До сих пор вся работа велась с холстом Matplotlib. График в окне при этом не изменялся. Чтобы обновить график в окне программы, нужно вызвать метод `draw()` Qt-виджета.

```
fc.draw()
```

Действия, описанные выше, в ходе работы программы могут выполняться многократно. Поэтому целесообразно оформить их в виде отдельной функции. Пример такой функции:

```
def plot_signal(freq, phase=0):
    for line in ax.lines:
        line.remove() # Удаление старых данных

    t = np.linspace(0, 10, 1000)
    signal = np.sin(2*np.pi*freq*t + phase)

    ax.plot(t, signal, label="Сигнал", color="green")
    ax.legend(loc=1) # Добавление легенды
    plt.tight_layout()
    fc.draw()
```

11.4 Обновление графика

Необходимость обновления графика обычно возникает при изменении входных данных (в нашем примере это частота и фаза сигнала). Данные могут изменяться пользователем через графический интерфейс приложения. Ниже приведён пример простейшего набора элементов пользовательского интерфейса, которые могут

использоваться для установки параметров сигнала.

Добавление текста:

```
label_ps = qt.QLabel(main_window)      # Привязка к окну
label_ps.setText("Параметры сигнала") # Установка текста
label_ps.move(440, 20)                 # Положение в окне
```

Добавление поля для ввода:

```
input_f = qt.QLineEdit(main_window)    # Привязка к окну
input_f.move(510, 50)                  # Положение в окне
input_f.resize(60, 20)                 # Размер поля
input_f.setText("0.2")                 # Значение по умолчанию
```

Добавление кнопки:

```
button = qt.QPushButton("Применить", main_window)
button.move(440, 110)                  # Положение в окне
button.resize(130, 25)                 # Размер кнопки
```

Чтобы параметры, вводимые в текстовые поля, при нажатии на кнопку применялись и приводили к перестроению графика, нужно:

1. Написать функцию-обработчик нажатия кнопки. Эта функция должна запускаться при каждом нажатии. Пример:

   ```
   def button_click_handler():
       f = float(input_f.text())
       plot_signal(f, 0)
   ```

2. Связать обработчик с соответствующим событием кнопки.

   ```
   button.clicked.connect(button_click_handler)
   ```

На рисунке 31 изображён пример окна, которое можно создать с применением средств, рассмотренных в данном разделе.

Стоит отметить, что при создании сложных интерфейсов средствами Qt иногда проще использовать Qt Designer.[15]

11.5 Автоматическое обновление графика

Иногда требуется периодически производить обновление данных без участия пользователя. Это можно сделать с помощью таймера:

1. Импортировать модуль QtCore.

   ```
   import PyQt5.QtCore as qtc
   ```

[15] https://doc.qt.io/qt-5/qtdesigner-manual.html

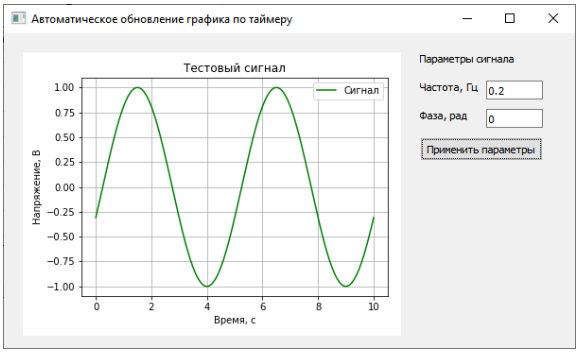


Рис. 31: Пример простого графического интерфейса, созданного средствами Qt

2. Объявить и инициализировать глобальные переменные, которые будут использоваться в обработчике таймера.

```
f = 0.5 # Частота сигнала, Гц
phi = 0 # Фаза сигнала, рад
```

3. Написать обработчик, который будет периодически вызываться по таймеру.

```
def replot_on_timer_handler():
    global f, phi
    phi = phi + np.pi/10
    plot_signal(f, phi)
```

4. Создать объект таймера, настроить его на непрерывную работу, назначить обработчик и запустить с требуемым периодом:

```
timer = qtc.QTimer(main_window)
timer.setSingleShot(False)
timer.timeout.connect(replot_on_timer_handler)
timer_period = 100 # Период, мс
timer.start(timer_period)
```

11.6 Задание для самостоятельного выполнения

Написать программу с графическим интерфейсом для построения и сохранения графиков. Программа должна читать данные для построения графика из текстового файла. Данные хранятся в файле в виде двух столбцов чисел (x и y), разделённых пробелами. При желании можно использовать другой формат файлов.

Графический интерфейс программы должен позволять:

- Просматривать график, построенный на основе загруженных из файла данных.
- Задавать имя файла для загрузки данных.
- Изменять заголовок графика.
- Изменять подписи осей.
- Сохранять построенный график в виде изображения.

Для сохранения графика в виде изображения можно использовать метод `savefig()` объекта класса `matplotlib.pyplot.Figure`.

Пример файла, содержащего входные данные:

```
1 1
1.9 4.2
3.1 6.1
```

12 Трёхмерная графика средствами пакета VPython

Бывает удобно представить результаты численного моделирования в виде анимированной трёхмерной модели некоторого набора физических объектов. Подобные демонстрации часто оказываются зрелищными и позволяют привлечь внимание к проекту.

Одним из простых инструментов для создания и программирования трёхмерных моделей с использованием языка Python является пакет Visual Python (сокращённо VPython), который в ходе развития проекта получил второе имя — GlowScript.[16] Стоит отметить, что VPython — это упрощённый инструмент, который позволяет быстро и просто создавать трёхмерные модели, но за счёт своей простоты обладает ограниченным функционалом.

Если для решения вашей задачи требуется профессиональный инструмент для программирования трёхмерной графики, рекомендуется использовать OpenGL с привязкой PyOpenGL для языка Python. Этот инструмент обладает значительно более широким набором возможностей по сравнению с VPython, но в свою очередь, требует гораздо больше времени для освоения.

12.1 Установка VPython

Если планируется использовать VPython в обычных программах, установить пакет можно с помощью пакетного менеджера pip

[16] https://www.glowscript.org/docs/VPythonDocs/index.html

(`pip install vpython`) или с помощью пакетного менеджера conda (`conda install -c vpython vpython`). Если планируется использовать VPython в тетрадях Jupyter, рекомендуется установить пакет IVisual[17]: (`pip install ivisual`). В этом случае для запуска последующих примеров вместо `import vpython as vp` нужно писать `import ivisual as vp`.

Зарегистрировавшись на официальном сайте[18], можно получить доступ к онлайн-среде разработки, которая позволяет создавать программы с использованием VPython без установки какого-либо программного обеспечения на свой ПК. Кроме того, сайт позволяет экспортировать программы в JavaScript для встраивания в веб-страницы.

При использовании IDE Spider для работы с VPython могут возникать дополнительные сложности. Способы решения проблем можно найти на официальном сайте в разделе Using the vpython module outside a Jupyter notebook (Python 3.5.3 or later).[19]

12.2 Создание сцены и простейших фигур

Пример создания трёхмерной сцены средствами VPython:

```
import vpython as vp

scene = vp.canvas()                  # Получение экземпляра сцены
scene.background = vp.color.white    # Установка цвета фона
s = vp.sphere()                      # Создание объекта (сферы)
```

При запуске этого фрагмента кода запустится браузер с возможностью просмотра созданной трёхмерной сцены.

Чтобы осуществлять обработку действий пользователя после завершения выполнения основной части программы, VPython «зависает» во внутреннем цикле. Программа должна завершаться автоматически после закрытия соответствующей вкладки (или окна браузера). Но это происходит не всегда. Для завершения работы сервера рекомендуется использовать следующий фрагмент кода:

```
from vpython.no_notebook import stop_server
stop_server()
```

Но, к сожалению, и этот способ не всегда работает достаточ-

[17] https://pypi.org/project/IVisual/
[18] https://www.glowscript.org
[19] https://www.vpython.org/presentation2018/install.html

но быстро. Самым простым и быстрым способом завершения программы, разработанной с использованием VPython, является закрытие терминала. Для быстрого повторного запуска можно создать файл `run.bat` (или `run.sh`, если вы используете Linux), содержащий строку запуска:

```
python main.py
```

При выполнении кода в тетради Jupyter, результат будет отображаться под ячейкой, в которой производится создание экземпляра объекта сцены (см. рисунок 32). При дальнейших изменениях параметров сцены и манипуляциях с объектами на ней, будет происходить обновление сцены. Новых сцен под ячейками ниже создаваться не будет.

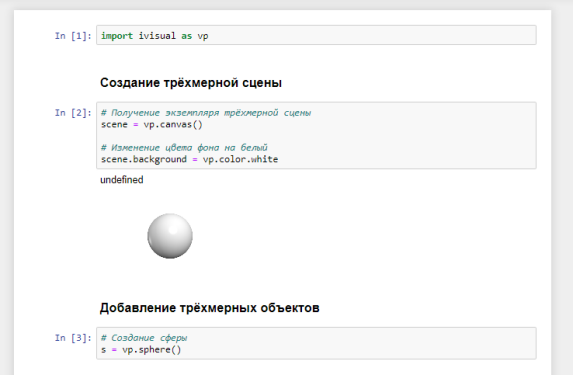


Рис. 32: Трёхмерная сцена VPython, встроенная в тетрадь Jupyter

Перемещение виртуальной камеры осуществляется следующим образом:

- Приближение/отдаление осуществляется с помощью колёсика мыши.
- Поворот камеры вокруг центра сцены происходит при перемещении мыши с нажатой левой кнопкой, а также нажатой клавишей *Ctrl* на клавиатуре.
- Перемещение вдоль сцены происходит при перемещении мыши с нажатой левой кнопкой, а также нажатой клавишей *Shift* на клавиатуре.

Действия, приведённые выше, по умолчанию приводят к перемещениям именно камеры. Положение объектов в сцене не изменяется.

12.3 Работа с трёхмерными объектами

VPython содержит набор классов для создания простых трёхмерных объектов: сфер, параллелепипедов, цилиндров... Создание трёхмерных объектов осуществляется путём создания экземпляров соответствующих классов. Обычно при создании объекта задаются координаты его центра, координаты вектора нормали и другие параметры. При необходимости все эти параметры в дальнейшем можно изменить, обращаясь к соответствующим полям. Пример добавления объектов в сцену и изменения их свойств:

```
import vpython as vp
scene = vp.canvas()
scene.background = vp.color.white

# Добавление объектов
ball = vp.sphere(pos=vp.vector(0, 0, 2),
                 radius=0.5,
                 color=vp.color.blue)
floor = vp.box(pos=vp.vector(0, 0, -0.1),
               size=vp.vector(4, 4, 0.1))

# Изменение свойств объекта путём обращения к его полям
floor.color=vp.color.green
```

Объекты появляются в сцене сразу же после создания. Изменение параметров объектов, уже размещённых в сцене, также происходит сразу после обновления соответствующих полей.

Практически все параметры, имеющие 3 координаты, должны передаваться в виде объектов класса `vpython.vector`.

Для удаления объекта, нужно выполнить 2 действия:

1. Удалить объект из сцены, установив значение `False` свойству `visible`.
2. Удалить объект из памяти с помощью оператора `del`.

Пример:

```
ball.visible = False
del ball
```

Если выполнить только первое действие, объект исчезнет со сцены, но останется в памяти. Если выполнить только второе действие, объект будет удалён из памяти, но продолжит отображаться в сцене. Поскольку самого объекта уже не будет, убрать объект из сцены или изменить его положение уже не получится.

Для удобства полной очистки сцены можно определить функ-

цию, удаляющую все доступные объекты:

```
def clear_canvas(canvas):
    for obj in canvas.objects:
        obj.visible = False
        del obj
```

Пример вызова:

```
clear_canvas(scene)
```

12.4 Координатная сетка

Положение объектов задаётся в трёхмерной декартовой системе координат. Бывает удобно отобразить в сцене начало координат и направление осей. Сделать это можно следующим образом:

```
pointer_x = vp.arrow(pos=vp.vector(0, 0, 0),
                     axis=vp.vector(1, 0, 0),
                     shaftwidth=0.1,
                     color=vp.color.red)
pointer_y = vp.arrow(pos=vp.vector(0, 0, 0),
                     axis=vp.vector(0, 1, 0),
                     shaftwidth=0.1,
                     color=vp.color.green)
pointer_z = vp.arrow(pos=vp.vector(0, 0, 0),
                     axis=vp.vector(0, 0, 1),
                     shaftwidth=0.1,
                     color=vp.color.blue)
```

В результате выполнения данного фрагмента кода в начале координат появятся 3 стрелки, указывающие направления осей.

12.5 Перемещение объектов

Перемещать объекты параллельным переносом можно, изменяя поле `pos`:

```
ball.pos = vp.vector(1, 1, 1)
```

Поворот объектов осуществляется с помощью метода `rotate()`:

```
floor.rotate(angle=vp.pi/2., axis=vp.vector(1, 0, 0))
```

Другой способ поворота объектов — указание координат вектора нормали:

```
floor.up = vp.vector(0, 1, 1)
```

12.6 Управление внешним видом объектов

Цвет объектов в VPython можно задать двумя способами.

1. Как готовый объект из модуля `vp.color`.

```
ball.color = vp.color.green
```

2. Как произвольный вектор в пространстве RGB. Каждая из координат вектора должна находиться в диапазоне [0, 1].

```
ball.color = vp.vector(0.3, 0.8, 0.2)
```

На объекты можно накладывать текстуры. В качестве текстур можно использовать произвольные изображения или воспользоваться одной из стандартных текстур.[20]

```
ball = vp.sphere(pos=vp.vector(0, 0, 0.5),
                 radius=0.5,
                 texture="ball_texture.jpg")
floor = vp.box(pos=vp.vector(0, 0, -0.1),
               size=vp.vector(4, 4, 0.1),
               texture="floor_texture.jpg")
ball_2 = vp.sphere(pos=vp.vector(0, 1.5, 0.5),
                   radius=0.5,
                   texture= vp.textures.earth)
```

Примеры использования текстур изображены на рисунке 33.

Рис. 33: Примеры использования текстур

Для более естественного отображения объектов можно задать степень глянца (поле `shininess`), которая изменяется в пределах от 0 до 1:

```
floor.shininess = 0.5
```

[20] https://www.glowscript.org/#/user/GlowScriptDemos/folder/Examples/program/Textures-VPython

Пример отображения текстур при изменении степени глянца изображён на рисунке 34.

Рис. 34: Вид текстур при разных значениях степени глянца (shininess)

В трёхмерной сцене можно создавать не только стандартные объекты, но и подписи. Пример подписи изображён на рисунке 35.

```
h_label = vp.label(pos=vp.vector(0.5, 0, 0), text="z = 0")
```

Рис. 35: Пример подписи в трёхмерной сцене

12.7 Элементы графического интерфейса

VPython позволяет не только создавать и изменять трёхмерные объекты, но и создавать относительно простые (по сравнению с Qt) пользовательские интерфейсы. Чтобы добавить элемент графического интерфейса, нужно:

1. Написать функцию-обработчик, которая должна запускаться при взаимодействии пользователя с элементом. Функция должна принимать один аргумент. При вызове в обработчик будет передан объект элемента, изменение которого привело к вызову обработчика.
2. Создать объект элемента графического интерфейса.
3. Связать функцию-обработчик с элементом графического интерфейса. Обработчик назначается полю `bind` и должен быть явно задан в момент создания объекта. В VPython обработчики можно назначить только один раз (при создании объекта) и в дальнейшем изменять нельзя.

При работе с текстовыми полями важно учесть следующие особенности. Значения текстовых полей обновляются при нажатии на клавишу `Enter`. По умолчанию в текстовых полях включена проверка того, что вводимый текст является числом. Если требуется

вводить не числа, а строки, при создании текстового поля атрибуту `type` нужно установить значение `string`. Пример трёхмерной сцены с элементами графического интерфейса пользователя:

```
import vpython as vp
import numpy as np

scene = vp.canvas()
scene.background = vp.color.white
ball = vp.sphere(pos=vp.vector(0, 0, 0.2),
                 radius=0.2,
                 color=vp.vector(0.1, 0.8, 0.2))
floor = vp.box(pos=vp.vector(0, 0, -0.1),
               size=vp.vector(1, 1, 0.1))

def textfield_handler(tf):
    x = float(tf.text)
    ball.pos = vp.vector(x, 0, 0.2)

scene.append_to_caption("\nПоложение шара по координате X."\
                        "Введите число и нажмите Enter.\n")
textfield = vp.winput(bind=textfield_handler, text=0)
```

Результат выполнения данного кода изображён на рисунке 36.

Рис. 36: Пример простого графического интерфейса пользователя

Помимо текстовых полей в VPython можно создавать:

- Кнопки

```
reset_z_button = vp.button(bind = button_handler,
                           text="Применить")
```

- Выпадающие списки

```
shininess_menu = vp.menu(choices=["1", "2", "3"],
                         bind=menu_handler)
```

- Ползунки

```
z_slider = vp.slider(bind=slider_handler, min=0, max=2)
```

12.8 Анимация

Анимация трёхмерных объектов осуществляется путём последовательного изменения положений и других свойств объектов в сцене. Для корректной обработки событий задержку между последовательными изменениями состояний нужно добавлять с помощью метода `vpython.sleep()`. Время задаётся в секундах. Пример кода для создания анимированной трёхмерной сцены:

```
import vpython as vp

scene = vp.canvas()
scene.background = vp.color.white

ball = vp.sphere(pos=vp.vector(0, 0, 0.1),
                 radius=0.1,
                 color=vp.vector(0.1, 0.1, 0.7))

h_label = vp.label(pos=vp.vector(0.25, 0, 0),
                   text="z = 0")
floor_h = 0.005
floor = vp.box(pos=vp.vector(0, 0, -floor_h),
               size=vp.vector(1, 1, floor_h))

def ball_set_z(z):
    ball.pos = vp.vector(0, 0, z + ball.radius)
    h_label.pos = vp.vector(0.25, 0, z)
    h_label.text = "z = {:1.1f} м".format(z)

while True:
    # Модель: z = v_0 * t - g * t^2 / 2
    g = 9.8 # м/с^2
    v_0 = 4 # м/с
    t = 0
    t_max = 2 * v_0 / g
    while t < t_max:
        z = v_0 * t - g * t ** 2 / 2
        ball_set_z(z)
        t = t + 0.005
        vp.sleep(0.02)
```

12.9 Задание для самостоятельного выполнения

Требуется создать анимированную трёхмерную модель процесса стрельбы из пушки. Пушка находится на уровне земли ($z = 0$) и осуществляет выстрелы в заданном направлении с фиксированной начальной скоростью ядра. В процессе полёта скорость ядра в горизонтальной плоскости постоянна (сопротивлением воздуха пренебречь). В вертикальном направлении ядро движется с ускорением под действием силы тяжести ($g = 9.8$ м/с^2). При падении на землю ($z = 0$) ядро должно останавливаться.

Модель должна предусматривать возможность изменения направления вектора начальной скорости ядра в вертикальной плоскости (угол, под которым осуществляется выстрел).

13 Язык HTML для описания содержимого веб-страниц

Для описания содержимого веб-страниц (в частности, страниц, которые мы видим на сайтах в интернете) используется язык форматирования HTML (*HyperText Markup Language*). Данный язык позволяет описывать элементы веб-страниц, их внешний вид и взаимное расположение. HTML не является языком программирования, поскольку он не описывает последовательности действий. Он описывает только внешний вид документа (аналогично LaTeX или Markdown), и поэтому он является языком разметки.

HTML описывает структуру документа посредством тегов — ключевых слов, заключенных в угловые скобки. Большая часть тегов является парными: элемент страницы заключается между парой тегов (открывающим и закрывающим), оба тега имеют одинаковое имя, но закрывающий тег начинается с символа /. Пример:

```
<title>Заголовок страницы</title>
```

Элементы HTML могут вкладываться друг в друга, что соответствует вложенности элементов документа. Например, гиперссылка может находиться внутри абзаца. Поэтому тег <a>, обозначающий гиперссылку, должен находиться внутри тега <p>, обозначающего абзац:

```
<p>Наиболее популярными в России поисковыми системами
являются <a href="https://www.google.com">Google</a>
и <a href="https://yandex.ru">Яндекс</a>.</p>
```

Существуют и одиночные теги, которые не имеют закрывающего тега. Чтобы показать, что тег одиночный, перед символом > обычно ставят символ /. Пример: `<br />` — тег, обозначающий перевода строки.

Теги могут иметь именованные параметры, которые перечисляются внутри открывающего тега после его имени, но до символа >. Примером такого тега является тег `img` для вставки изображения из файла, заданного параметром `src`, с указанием краткого описания (параметр `alt`).

```
<img src="myTestPicture.jpg" alt="Изображение для теста" />
```

Комментарии в HTML оформляются с помощью тега `<!-- -->`.

```
<!-- Это комментарий. Данный текст
     не будет интерпретироваться браузером -->
```

13.1 Структура HTML-документа

В начале любого HTML-документа следует поместить служебный тег `<!DOCTYPE html>`, который сообщает браузеру, что дальнейшее содержимое документа нужно интерпретировать как HTML. Всё содержимое документа должно находиться между парой тегов `<html></html>`.

Документы HTML состоят из двух частей, которые определяются внутри соответствующих тегов:

- *head* — заголовок документа, который содержит его название, краткое описание, ключевые слова, информацию о кодировке, используемых стилях и другую служебную информацию.
- *body* — тело документа, в котором содержится непосредственно описание структуры и свойств его элементов.

Пример HTML-документа:

```
<!DOCTYPE html>
<html>
   <head> <!-- Заголовок документа -->
      <meta charset="utf-8" />
      <title>Тестовый документ</title>
   </head>
   <body> <!-- Тело документа -->
      Пожалуй, это наиболее простой HTML-документ,
      который можно придумать.
   </body>
</html>
```

Если сохранить код, приведённый выше, в текстовый файл с расширением `.html` (например, `test.html`), а затем открыть в браузере (программе, которая обычно используется для просмотра веб-страниц: Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge, Opera,...), результат должен выглядеть примерно так, как показано на рисунке 37.

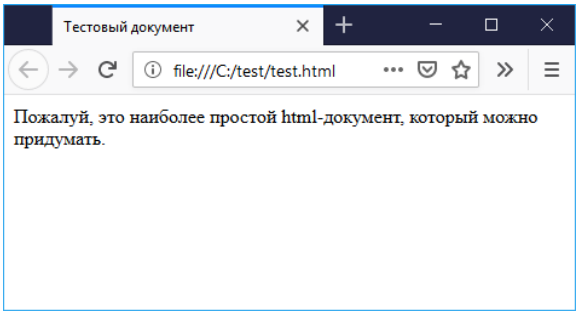


Рис. 37: Пример простой HTML-страницы

13.2 Базовые элементы HTML-документов

Заголовки

Заголовки определяются парными тегами вида `<hn></hn>`, где `n` — уровень заголовка. `h` — сокращение от *header*.

```
<h1>Заголовок первого уровня</h1>
<h2>Заголовок второго уровня</h2>
<h3>Заголовок третьего уровня</h3>
<h4>Заголовок четвёртого уровня</h4>
```

Абзацы и переносы строк

Абзацы выделяются парными тегами `<p></p>` (от английского *paragraph*). Тег `<p>` может иметь атрибут `align`, отвечающий за выравнивание текста и принимающий одно из четырёх возможных значений: `"left"` (по левому краю), `"right"` (по правому краю), `"center"` (по центру), `"justify"` (по двум сторонам). Для переноса строк используется одиночный тег `<br />`.

```
<p align="justify">Абзац будет выровнен по двум сторонам.
<br />А это предложение начнётся с новой строки.</p>
```

Ссылки и изображения

Ссылки выделяются с помощью парных тегов `<a></a>`. URL страницы, на которую нужно сослаться, задаётся атрибутом `href`.

Для вставки изображений используется одиночный тег `<img />`. Путь к файлу с изображением указывается в атрибуте `src`, также обычно добавляется атрибут `alt`, с помощью которого задаётся текст с кратким описанием изображения. Этот текст будет демонстрироваться в случае, если файл с изображением окажется недоступен.

Пример использования описанных выше элементов:

```
<h1>ВДНХ</h1>
<p align="justify">Выставка Достижений Народного Хозяйства
(ВДНХ) - это второй по величине выставочный комплекс
в Москве. Подробную информацию о всех мероприятиях,
происходящих на ВДНХ, можно найти на
<a href="https://vdnh.ru">официальном сайте</a>.</p>

<h4>Главный вход ВДНХ</h4>
<img src="vdnh.jpg" alt="Главный вход ВДНХ" />
```

Вид получившейся HTML-страницы при отображении в браузере показан на рисунке 38.

Чтобы изображение отображалось корректно, файл `vdnh.jpg` должен располагаться в том же каталоге, что и HTML-файл с описанием страницы.

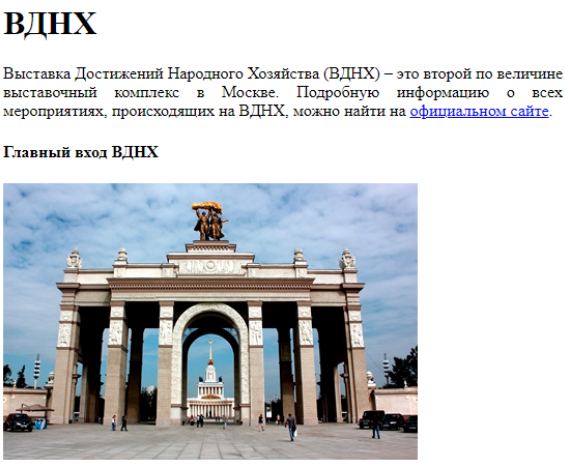


Рис. 38: Пример HTML-страницы с различными элементами форматирования

Таблицы

Для добавления таблиц используются теги `<table></table>`. Строки внутри таблицы определяются тегами `<tr></tr>`. Ячейки в строке, в свою очередь, обозначаются тегами `<td></td>`.

```
<table border="1" cellpadding="5">
    <tr>
        <td>Класс частиц</td>
        <td>Спин</td>
        <td>Статистика</td>
        <td>Дополнительно</td>
    </tr>
    <tr>
        <td>Фермионы</td>
        <td>Полуцелый</td>
        <td>Статистика Ферми-Дирака</td>
        <td>Согласно принципу Паули два одинаковых фермиона
            не могут находиться в одном квантовом состоянии.
       </td>
    </tr>
    <tr>
        <td>Бозоны</td>
        <td>Целый</td>
        <td>Статистика Бозе-Эйнштейна</td>
        <td>Бозоны, охлаждённые до температур
            близких к абсолютному нулю,
            занимают низшее квантовое состояние.
            Этот процесс называют конденсацией Бозе-Эйнштейна.
        </td>
    </tr>
</table>
```

На рисунке 39 показано, как такая таблица будет выглядеть в браузере.

Класс частиц	Спин	Статистика	Дополнительно
Фермионы	Полуцелый	Статистика Ферми-Дирака	Согласно принципу Паули два одинаковых фермиона не могут находиться в одном квантовом состоянии.
Бозоны	Целый	Статистика Бозе-Эйнштейна	Бозоны, охлаждённые до температур близких к абсолютному нулю, занимают низшее квантовое состояние. Этот процесс называют конденсацией Бозе-Эйнштейна.

Рис. 39: Пример отображения HTML-таблиц

Списки

В HTML предусмотрены теги для создания нумерованных и ненумерованных списков. Нумерованные списки задаются внутри тегов `<ol></ol>` (*ordered list*), ненумерованные списки — внутри тегов `<ul></ul>` (*unordered list*). Отдельные элементы внутри списка в обоих случаях задаются тегами `<li></li>` (*list item*).

Пример оформления нумерованного списка:

```
<h4>Нумерованный список</h4>
<ol>
        <li>Элемент 1</li>
        <li>Элемент 2</li>
        <li>Элемент 3</li>
</ol>
```

Пример оформления ненумерованного списка:

```
<h4>Ненумерованный список</h4>
<ul>
        <li>Элемент 1</li>
        <li>Элемент 2</li>
        <li>Элемент 3</li>
</ul>
```

Примеры отображения нумерованного и ненумерованного списка в браузере изображены на рисунке 40.

Нумерованный список

1. Элемент 1
2. Элемент 2
3. Элемент 3

Ненумерованный список

- Элемент 1
- Элемент 2
- Элемент 3

Рис. 40: Пример отображения нумерованного и ненумерованного списка

13.3 CSS

Cascading Style Sheets (CSS) — это наборы стилей, которые позволяют описывать внешний вид тех или иных объектов HTML-документа. Стили определяются между тегов `<style></style>` в заголовке (head) HTML-документа либо в отдельных файлах с расширением .css.

Чтобы применить стиль CSS к объекту HTML-документа, нужно сначала определить новый класс в таблице стилей, а затем с

помощью атрибута `class` указать, что созданный класс со стилями нужно применять к соответствующему объекту HTML-документа.

Пример определения стиля, позволяющего изменить цвет текста на красный (код, приведённый ниже, нужно поместить между тегами `<head></head>`).

```
<style>
    /* Определение класса red_text */
    *.red_text {
        color:#FF0000; /* Явное указание цвета текста */
    }
</style>
```

Пример применения созданного стиля к фрагменту текста:

```
<span class="red_text">Красный текст</span>
```

Здесь были рассмотрены только базовые возможности языков HTML и CSS. Данные языки являются очень гибкими и предоставляют практически безграничные возможности для оформления веб-страниц.

14 Создание простых веб-серверов средствами Python

HTML-документы, как обычные файлы, можно просматривать в браузере на локальном ПК, но чтобы получить доступ к этим файлам с других устройств, требуется специальная программа — *HTTP-сервер*. Когда пользователь вводит в браузере адрес сайта (или ip-адрес устройства) и нажимает Enter, браузер отправляет на соответствующий узел сети HTTP-запрос. Такой запрос содержит информацию о том, что определённая программа (например, интернет браузер) запрашивает HTML-документ с определённого узла (например, `www.yandex.ru`) и по возможности просит предоставить информацию на определённом языке (например, русском).

HTTP-сервера принимают подобные запросы, обрабатывают их и возвращают клиенту (в роли которого в данном случае выступает браузер) запрашиваемые данные. В общем случае такие сервера могут предоставлять не только HTML-документы, но также изображения, музыку, PDF-документы и любые другие файлы.

HTTP-сервера могут отправить пользователю данные только в качестве ответа на его запрос. Отправлять какие-либо данные без

запроса клиента сервер не может. Классические HTTP-сервера не хранят никакой информации о клиентах между запросами. Поэтому каждый запрос с точки зрения сервера посылается новым клиентом. Если клиент хочет себя как-то идентифицировать, он должен отправить соответствующие данные в запросе.

На одном ПК может работать несколько серверных приложений (не обязательно HTTP-серверов), которые должны отвечать на запросы, приходящие по сети. Поэтому для адресации в пределах ПК используются порты. Порт определяется цифрой, которая обычно указывается после IP адреса и отделяется от него двоеточием. Разные серверные приложения могут работать на разных портах на одном ПК с единым IP-адресом. В зависимости от переданного значения порта запросы будут направляться тому серверному приложению, которое зарегистрировано на соответствующем порту. Например, сервер тетрадей Jupyter обычно использует порт `8888`.

Существует список стандартных tcp/ip портов.[21] Если ПК получает запрос без указания порта, запрос перенаправляется приложению, которое использует стандартный для данного протокола порт. Стандартным портом для протокола HTTP является порт 80. Чтобы пользователям не приходилось каждый раз указывать после URL сайта порт в явном виде, нужно запускать HTTP-сервера на порту 80 или настраивать перенаправление портов.

Стоит отметить, что большинство современных сайтов передают данные по протоколу HTTPS. Этот протокол представляет собой расширение HTTP, которое подразумевает шифрование передаваемых данных и дополнительные процедуры проверки подлинности ресурсов. Стандартный порт для HTTPS — 443.

14.1 Статический HTTP-сервер

Есть целый ряд пакетов для языка Python, которые позволяют создавать HTTP-сервера: Flask, Django, aiohttp... Но для первого знакомства с веб-разработкой имеет смысл выбрать наиболее простой пакет. Поэтому мы будем использовать пакет http, входящий в стандартную библиотеку.

Статическими серверами называют сервера, предназначенные только для передачи запрашиваемых файлов и не подразумевающие какую-либо «умную» обработку запросов. Код простого стати-

[21] https://ru.wikipedia.org/wiki/Список_портов_TCP_и_UDP

ческого сервера, написанного с использованием пакета http, занимает всего 4 строки:

```
from http.server import (HTTPServer,
                         SimpleHTTPRequestHandler)

srv_address = ("", 80) # 80 - стандарный порт HTTP
httpd = HTTPServer(srv_address, SimpleHTTPRequestHandler)

# Запуск сервера
# Сервер синхронный, поэтому тут произойдёт блокировка
httpd.serve_forever()
```

После запуска скрипта можно открыть браузер и написать в адресной строке `localhost` или `127.0.0.1` (адреса, которые позволяют устройству обращаться к самому себе по сети). В браузере должна открыться страница со списком файлов, находящихся в каталоге сервера (см. рисунок 41).

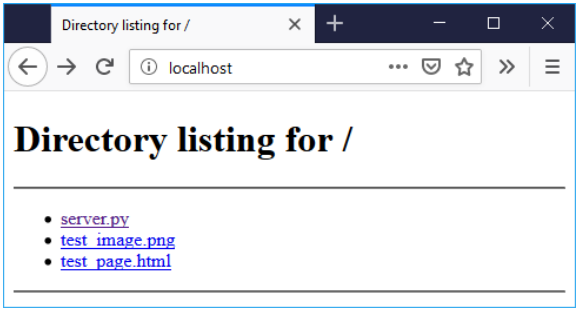


Рис. 41: Пример отображения списка файлов, доступных на статическом сервере

При попытке пройти по ссылке, соответствующей одному из файлов, в зависимости от возможностей вашего браузера файл либо откроется непосредственно в окне программы, либо загрузится на ваш ПК.

Если создать в каталоге файл под названием `index.html`, при переходе на `localhost` список всех файлов в текущем каталоге отображаться не будет. Вместо этого в браузере сразу откроется страница, описанная в файле `index.html`.

Внутри каталога с сервером можно создавать вложенные каталоги. Доступ к содержимому этих каталогов можно получить, введя в браузере адрес вида `localhost/<имя вложенного каталога>`.

В случае, если при создании сервера был указан порт, отлич-

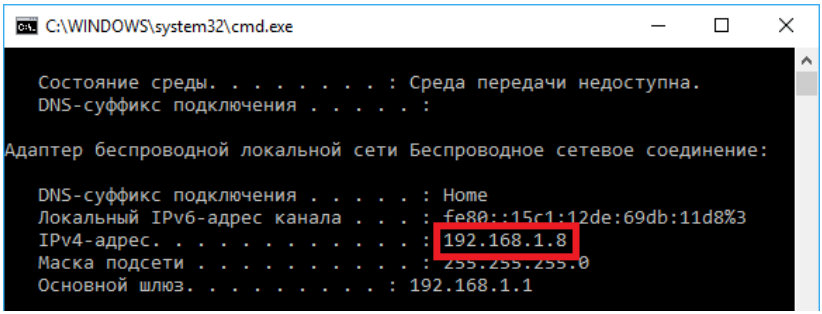


Рис. 42: Определение IP адреса с помощью команды ipconfig

ный от 80, при обращении к серверу после localhost всегда нужно указывать номер порта. Пример: `localhost:8080`.

Для обращения к запущенному HTTP-серверу с другого устройства, находящегося в той же сети, нужно знать IP-адрес ПК, на котором запущено серверное приложение. Этот IP-адрес нужно указать в адресной строке браузера вместо `localhost`. Узнать IP-адрес ПК под управлением ОС Windows можно с помощью консольной команды `ipconfig` (см. рисунок 42). В Linux есть аналогичные команды: `ifconfig` и `ip a`.

Чтобы получить доступ к HTTP-серверу через интернет, нужно, во-первых, знать IP адрес в интернете (а не в локальной сети), а во-вторых, настроить роутер сети, в которой находится сервер, таким образом, чтобы порты, используемые сервером, были доступны из внешней сети.

14.2 CGI-сервер

CGI-сервера отличаются от статических серверов тем, что в случае, если запрашиваемый файл является исполняемым, они не возвращают клиенту сам файл, а запускают файл на выполнение и возвращают клиенту вывод запущенного файла.

По умолчанию обработчик запросов `CGIHTTPRequestHandler` из пакета http запускает только файлы, расположенные в каталоге `cgi-bin`. При необходимости можно указать серверу и другие каталоги, содержащие файлы, которые нужно запускать при обработке HTTP-запросов. Для этого нужно задать список директорий через поле `CGIHTTPRequestHandler.cgi_directories`. Код CGI-сервера не сложнее кода статического HTTP-сервера.

```
from http.server import (HTTPServer,
                         CGIHTTPRequestHandler)
# Создание сервера
# Настройка запуска CGI сценариев из корневой директории
CGIHTTPRequestHandler.cgi_directories = ["/"]
server_address = ("", 80) # 80 - стандарный порт HTTP
httpd = HTTPServer(server_address, CGIHTTPRequestHandler)

# Запуск сервера (выполнение скрипта при этом блокируется)
httpd.serve_forever()
```

В случае возврата HTML-документов в первой строке ответа нужно сообщить браузеру, что далее будет отправлено содержимое в формате `text/html`. Поэтому первая строка ответа сервера должна иметь вид:

```
Content-type: text/html
```

Важно, чтобы после `text/html` был произведён перевод строки. В противном случае браузер неправильно воспримет переданный тип данных.

Если в каталоге с сервером создать файл `test_page.py` со следующим содержимым

```
import math

# Заголовок ответа. \n в конце очень важен
print("Content-type: text/html\n")

# Возврат полезного содержимого
print("pi * e = ", math.pi * math.e)
```

то при запросе страницы `http://localhost/test_page.py` через браузер, сервер выполнит данный сценарий и вернёт в окно браузера результат вычислений:

```
pi * e = 8.539734222673566
```

Для корректной работы этого и последующих примеров необходимо, чтобы во всех файлах использовалась кодировка UTF-8.

14.3 Шаблонизация при формировании ответов

Пример сценария-обработчика, приведённый в предыдущем разделе является не совсем корректным, поскольку он не отправляет клиенту даже минимального набора обязательных атрибутов HTML-документа. Кроме того, реальные веб-страницы содержат

гораздо больше различных элементов.

Печатать все необходимые теги и другие элементы веб-страницы с помощью функции `print()` — не самый удобный подход. Гораздо удобнее заранее подготовить шаблон HTML-страницы в виде отдельного файла, в котором будет всё, кроме непосредственно тех данных, которые генерируется динамически в процессе выполнения сценария-обработчика запроса. При получении запроса в сценарии будет осуществляться только подготовка соответствующих динамических данных, загрузка шаблон HTML-страницы из файла в виде строки, подстановка динамических данные и возврат клиенту заполненного шаблона.

Шаблонизацию можно организовать, например, с помощью метода `format()` стандартных строк языка Python. В этом случае места для подстановки динамических данных в шаблоне обозначаются фигурными скобками `{}`. Дополнительно в фигурных скобках можно указать имя поля. Пример: `{my_variable}`.

Рассмотрим пример шаблонизированной страницы. При обращении по адресу `http://localhost/random_add.py` сервер будет генерировать два случайных числа, вычислять их сумму, подставлять полученные числа в шаблон страницы и возвращать её пользователю.

Код шаблона (пусть для определённости этот код хранится в файле `random_add_template.html`):

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Сложение двух случайных чисел</title>
   </head>
   <body>
      <h1>Сложение двух случайных чисел</h1>
      <p>{random_num_1} + {random_num_2} = {random_sum}</p>
   </body>
</html>
```

Файл сценария-обработчика HTTP-запросов (пусть для определённости сценарий будет сохранён в файле `random_add.py`):

```
import numpy as np

# Загрузка шаблона из файла
with open("random_add_template.html", "r") as file:
    template = file.read()

# Подготовка данных
r_1 = np.random.random()
r_2 = np.random.random()
sum = r_1 + r_2

# Заполнение шаблона
page = template.format(random_num_1 = r_1,
                       random_num_2 = r_2,
                       random_sum = sum)

# Отправка заполненного шаблона клиенту
print("Content-type: text/html\n")
print(page)
```

Для корректной работы примера файлы `random_add_template.html` и `random_add.py` должны находиться в одном каталоге. Вид страницы, которая будет возвращаться сервером при обращении по адресу `http://localhost/test_page.py`, изображён на рисунке 43.

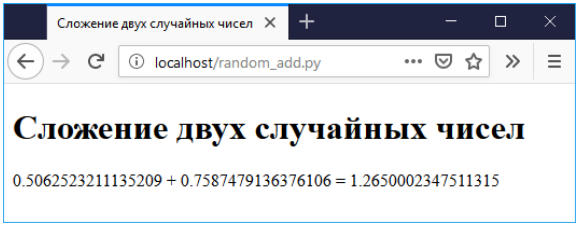


Рис. 43: Вид шаблонизированной страницы, осуществляющей генерацию двух случайных чисел и вычисление их суммы на сервере

14.4 Обработка данных HTML-форм

Для обработки данных HTML-форм нужно подключить модуль `cgi`. Объект с параметрами формы можно получить с помощью функции `cgi.FieldStorage()`. Значения конкретных полей можно получать с помощью метода `getfirst()` по их именам.

Рассмотрим обработку HTML-форм на примере страницы, которая позволяет вычислить сумму двух чисел, вводимых пользователем.

Файл шаблона `calculator_template.html`:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title>Простой калькулятор (сумматор)</title>
   </head>
   <body>
      <h1>Простой калькулятор (сумматор)</h1>
      <!-- calculator.py - обработчик формы -->
      <form action="calculator.py">
        <input type="text" value="{a_val}" name="a_in"> +
        <input type="text" value="{b_val}" name="b_in">
          = {result}<br />
        <input value="Вычислить" type="submit">
      </form>
   </body>
</html>
```

Код сценария-обработчика `calculator.py`:

```
import cgi

# Получение переданных через форму параметров
form = cgi.FieldStorage()
try:
    # Попытка получить значения полей формы
    a = int(form.getfirst("a_in", "0"))
    b = int(form.getfirst("b_in", "0"))
    result = a + b
except ValueError:
    # Обработка ошибок (если пользователь ввёл не число)
    a = form.getfirst("a")
    b = form.getfirst("b")
    result = "Data incorrect"

# Загрузка и заполнение шаблона
with open("calculator_template.html", "r") as file:
    template = file.read()
page = template.format(a_val=a, b_val=b, result=result)

# Отправка заполненного шаблона клиенту
print("Content-type: text/html\n")
print(page)
```

Результат запроса страницы `http://localhost/calculator.py` через браузер изображён на рисунке 44.

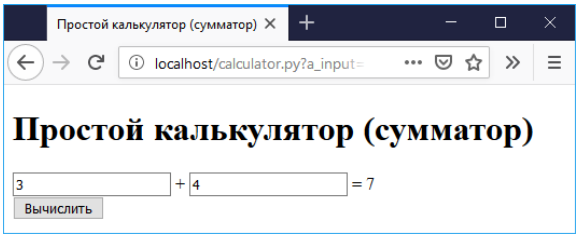


Рис. 44: Пример HTML-страницы с формой, позволяющей вычислить сумму двух чисел

14.5 Задание для самостоятельного выполнения

Написать HTTP-сервер, возвращающий HTML-страницу, содержащую информацию о текущем времени (часы, минуты и секунды). При обновлении страницы возвращаемое значение времени должно обновляться. Сервер должен формировать HTML-страницу на основе шаблона, хранящегося в отдельном файле.

Литература

[1] G. van Rossum and the Python development team. The Python Language Reference. Release 3.8.6.

[2] G. van Rossum and the Python development team. The Python Library Reference. Release 3.8.6.

[3] ipywidgets documentation 7.5.1.

[4] G. van Rossum, B. Warsaw, N. Coghlan. PEP 8 — Style Guide for Python Code

[5] SymPy Documentation by SymPy Development Team. Release 1.6.2

[6] W. McKinney and the Pandas Development Team. Pandas: powerful Python data analysis toolkit. Release 1.1.2

[7] NumPy Reference Guide by NumPy community. Release 1.17.0

[8] SciPy Reference Guide by SciPy community. Release 1.5.2

[9] Ильин В. А., Позняк Э. Г. Основы математического анализа: В 2х ч. Часть II: Учеб.: Для вузов. — 4-е изд. — М.: Физматлит, 2002.

[10] Воронцов Ю. И., Вятчанин С. П. Введение в радиофизику. М.: Физический факультет МГУ имени М.В. Ломоносова, 2016. — 204 стр.

[11] PyQt5 Reference Guide. v5.15.1

[12] J. Hunter, D. Dale, E. Firing, M. Droettboom and the matplotlib community. Matplotlib. Release 2.0.0

[13] Прохоренок Н. А. Python 3 и PyQt. Разработка приложений.

[14] HTML 5.2. W3C Recommendation, 14 December 2017

[15] Ташков П. Веб-мастеринг на 100%. HTML, CSS, JAVASCRIPT, PHP, CMS, AJAX, раскрутка. СПб: «Питер» 2010.

[16] Бизли Д. Python. Подробный справочник. — Пер. с англ. — СПб: Символ-Плюс, 2010.