Оглавление

Введение

Данное учебно-методическом пособие предназначено для организации лабораторных работ по семестровому курсу «Компьютерная геометрия и математическое моделирование» по специальности 02.03.01 «Математика и компьютерные науки». Целью лабораторных работ является практическое овладение аппаратом компьютерной геометрии. В связи с ограничением курса одним семестром, программа охватывает только двумерную геометрию и ограничивается лишь плоскими кривыми, в частности изучаются сплайны Эрмита, Ньютона, Лагранжа, Катмулла-Рома, кубические сплайны, а также кривые Безье.

В качестве основных инструментов для реализации методов компьютерной геометрии предлагается использовать язык `Python` в связке с библиотеками `NumPy`, `Matplotlib` и интерактивной оболочкой `Jupyter`. Три главы данного пособия посвящены краткому введению в основные возможности языка `Python` и вышеперечисленных библиотек.

Кратко опишем структуру методического пособия.

- В первой главе излагается необходимый минимум сведений о языке `Python`, который может пригодится для использования библиотеки `Matplotlib` и реализации алгоритмов компьютерной геометрии. Упор в изложении делается на многочисленные примеры, которые предлагается использовать для самостоятельной работы студентов.

- Вторая глава посвящена базовым возможностям библиотеки `NumPy` с упором на те ее возможности, которые могут найти применение в задачах компьютерной геометрии и в связке с библиотекой `Matplotlib`.

- Третья глава посвящена библиотеке `Matplotlib` и подробно описывает те ее средства, которые можно использовать для изображения плоских кривых и реализации алгоритмов компьютерной геометрии. Большой набор методов, предназначенных, например, для анализа данных и обработки сигналов, остался вне изложения.

- Четвертая глава является краткой справкой по математическому аппарату компьютерной геометрии и посвящена в основном различным сплайнам и кривым Безье.

- В конце каждой из первых трех глав приводятся по две лабораторные работы, выполнение которых позволит оценить уровень усвоения материала курса. В конце четвертой главы приведено шесть лабораторных работ, которые покрывают весь теоретический материал курса.

Следует добавить, что каждая глава снабжена списком контрольных вопросов, которые предназначены для самопроверки. Для ответа на некоторые из этих вопросов учащимся необходимо воспользоваться дополнительной литературой и официальной документацией.

Глава 1

Краткие сведения о языке программирования Python

В данном разделе излагаются сведения о языке программирования Python третьей версии. Многие средства языка изложены не будут. Также почти незатронутой останется обширная стандартная библиотека, функционал языка, касающийся объектно-ориентированного подхода к программированию и т.д. Упор будет сделан только на те сведения, которые, на наш взгляд, пригодятся для математических вычислений и работы с научными библиотеками NumPy, SciPy и особенно Matplotlib. Читателя, желающим изучить язык более подробно, можно порекомендовать книги [1, 2, 3, 4], а также официальную документацию [5], которая обладает обширным обучающим разделом.

1.1. Характеристика языка

Язык программирования Python [6, 5] является интерпретируемым языком общего назначения. Python обладает строгой динамической типизации. Динамичность означает, что тип переменной определяется в момент инициализации переменной, в свою очередь строгость типизации говорит о том, что автоматическое приведение типов за редким исключением не используется.

Язык отличается крайне компактным синтаксисом и малым количеством базовых инструкций. Широкая функциональность обеспечивается обширной стандартной библиотекой и не включена в ядро языка.

Язык Python обрел популярность в следующих основных областях.

- Как скриптовый язык для администрирования операционной системы и создания небольших системных и прикладных консольных утилит.
- В области сетевых сервисов и приложений.
- В области научных вычислений благодаря многочисленным библиотекам, таким как NumPy [7], SciPy [8], SymPy [9], Matplotlib [10], Numba [11] и многие другие.

Перечисленные ниши не исчерпывают всех областей применения языка, однако являются для него основными. Нас больше всего будет интересовать применения языка Python в области научных вычислений. В настоящее время Python составляет конкуренцию в этой области

таким известным коммерческим математическим пакетам как Matlab [12], Mathematica [13] и Maple [14, 15].

Заметим также, что в настоящее время существуют две версии языка Python — версия 2 и версия 3. Вторая версия не развивается и ее поддержка вскоре должна полностью прекратиться, однако она все еще пользуется популярностью. В данном пособии мы используем исключительно третью версию.

Следует подчеркнуть, что язык Python не стандартизирует интерпретатор и поэтому существует большое количество реализаций интерпретаторов Python. Стандартным интерпретатором является CPython [16], реализованный на языке C (C89 и частично C99). CPython преобразует код Python в байт-код, который потом выполняется на виртуальной машине. Все примеры данного пособия предполагают использвание именно CPython.

Упомяним также несколько альтернативных интерпретаторов.

- Jython [17] — интерпретатор реализованный на языке Java и Python. Транслирует Python-код в Java-байткод, что дает возможность испольнять созданные программы на виртуальных Java машинах (JVM).
- PyPy [18] — интерпретатор, реализован на Python (подмножество RPython — Restricted Python). Поддерживается синтаксис второй версии. Отличительной особенностью является улученная поддержка многопоточности.
- IronPython [19] — интерпретатор на C#. Транслирует Python-код в байт код для платформы .NET компании Microsoft.

1.2. Основные типы данных

1.2.1. Числовые типы данных и элементарные вычисления

В силу динамической природы языка Python для создания переменной достаточно присвоить ей некоторое значение. Интерпретатор самостоятельно определит, под какой тип данных следует выделить память. Например следующий код

```
1 j = 1
```

создаст переменную j типа int (целый тип). Заметим, что для выяснения типа переменной можно воспользоваться встроенной функцией type(). Рассмотрим несколько примеров, иллюстрирующих элементарные арифметические действия.

```
1 >>> 1 + 3**2 - (6 - 5)
2 9
3 >>> 0.1 + 0.2 + 0.3 + 1e4
4 10000.5
5 >>> print('Привет мир!')
```

Оператор ** позволяет возводить в степень числа любого типа. При совершении действий над числовыми переменными различного типа, происходит автоматическое приведении типа

к более общему. Встроенная функция `print()` позволяет выводить имнформацию в стандартный поток вывода (консоль).

Следующий код показывает основные типы данных, доступные в Python:

```
j = 2 # Целое число
a = 5.0 # Действительное число
A = 1.0 # Регистр имеет значение
y =  1.0 + 1.0j # Комплексное число
str1 = "Это некоторый текст"
str2 = 'Одинарные кавычки тоже допустимы'
print(j, a, A, y, str1, str2)
```

1.2.2. Строки и их методы

Строки в `Python` являются одним из базовых типов. Они реализованы как сложные объекты, имеющие функции-члены (методы) и поля (атрибуты). Для доступа к методам используется оператор точка `.`. Для получения полного списка атрибутов и методов можно использовать встроенную функцию `dir(object_name)`. Рассмотрим несколько методов строк.

```
>>> msg = "Здравствуйте!"
>>> msg.upper() # в верхний регистр
'ЗДРАВСТВУЙТЕ!'
>>> msg.lower() # в нижний регистр
'здравствуйте!'
>>> "CAPS LOCK".lower()
'caps lock'
```

Часто приходится разбивать строки на части, используя некоторый символ в качестве разделителя. Рассмотрим как это делается.

```
animals = "Заяц,Кролик,Гусь,Селезень"
l = animals.split(',') # создается список из 4 строк
print(l)
```

Тройные кавычки позволяют задавать большие куски текста (длинные строки), а метод `splitlines()` позволяет разбивать такой блок текста на строки:

```
block = """1,2,32
2,76,54
3,32,454
"""
s = block.splitlines()
print(s)
['1,2,32', '2,76,54', '3,32,454']
```

Для конкатенации строк служит оператор `+`:

```
>>>"Раз" + " и " + "два!"
'Раз и два!'
```

Однако гораздо более гибок метод `format()`, который служит для форматирования строк и похож на функцию `printf` языка `C`.

```
"a = {0}, a^2 = {1}".format(a, a**2)
```

Для конвертации других типов в строку, служит встроенная функция `str()`. Используя ее и оператор `+`, можно формировать сложные строки, содержащие значения любых переменных. Однако предпочтительнее для таких целей использовать метод `format()` так как он намного эффективней работает с памятью.

1.2.3. Логический тип и логические выражения

Зарезервированные слова `True` и `False` служат для обозначения лжи и истины.

```
X = True
Y = False
X is Y # False
X is X # True
Y is Y # True
6 < 4 # False
5 == 1 # False
```

Вхождение подстроки в строку можно проверять с помощью оператора `in`.

```
"След" in "Следствие" # True
"кость" in "плоскость" # True
```

Логические операторы конъюнкции, дизъюнкции, отрицания и строгой дизъюнкции реализованы в виде английских слов `and`, `or`, `not` и `xor`.

```
6 and 3 # True
6 or 0 # True
True is not False # True
```

Операции сравнения >, <, >= и <= можно комбинировать также, как в математических выражениях. Так, например корректно следующее выражение:

```
6 <= x < 3
```

1.3. Управление потоком выполнения

В синтаксисе языка `Python` нет открывающих и закрывающих конструкций вроде слов `begin` и `end` или скобок `{` и `}`. Для определения вложенности инструкций используются отступы. В качестве отступов можно применять табуляцию или четыре пробела. Официальная документация рекомендует использовать исключительно четыре пробела. Благодаря этому текст программы становится короче и читаемее, так как программист вынужден всегда делать отступы и иерархия управляющих конструкций всегда явно видна.

Рассмотрим оператор условного перехода `if-elif-else` и циклы `for` и `while`.

1.3.1. Оператор условного перехода if

Оператор условного перехода `if` заменяет как классический оператор `if`, так и оператор `switch/case`. Проиллюстрируем его применение на примере:

```
a = 4
if a == 4:
    print("Четыре")
elif a == 5:
    print("Пять")
elif 0 <= a < 4:
    print("Меньше четырех")
else:
    print("Что-то иное")
>>> 'Четыре'
```

Двоеточие после каждого условия и безусловного `else` обязательно. Для записи условия не требуются круглые скобки. Для повышения читаемости и наглядности можно сложные условия вычислять отдельно.

```
x = 1
y = 2
z = 3
p = 3
condition = (x < y < z < p)
if condition:
    print("Условие истинно")
else:
    print("Условие ложно")
>>> 'Условие ложно'
```

1.3.2. Циклы

Цикл `for` следует использовать для перебора элементов некоторой коллекции (множество, список, кортеж и т.д.). Рассмотрим следующий пример.

```
for i in range(1, 10, 1):
    print(i, end=' ')
>>> '1 2 3 4 5 6 7 8 9'
```

Здесь также используется двоеточие. Встроенная функция `range(start, stop, step)`, позволяет перебрать целые числа начиная от `start` и заканчивая `stop-1` с шагом `step`. Заметим также, что правая граница диапазона не входит в конечный набор чисел. Это поведение в работе с индексами характерно для языка и будет встречаться далее повсеместно.

В языке `Python` цикл `while` имеет стандартный синтаксис, поэтому мы ограничимся лишь одним примером.

```
x = 1.3
while x > 1:
    x = x - 1.1
```

1.4. Списки

В языке Python нет встроенных классических массивов, однако есть более гибкий тип данных — список (list). Список может содержать элементы различных типов, обеспечивает доступ к элементам по индексам, возможность добавлять и удалять элементы. Список Python универсален и может выполнять роль стека, очереди и двусторонней очереди.

Классические массивы отсутствуют в Python, однако этот недостаток устраняется библиотекой NumPy, которую мы очень кратко рассмотрим в конце этой главы.

1.4.1. Создание списка

Создается список посредством квадратных скобок [] или функции list().

```
>>> l = [] # Пустой список
>>> l = [6, 7, 8, 9, 10]
>>> print(l)
[6, 7, 8, 9, 10]
>>> [False, 1, "Привет"] # Список с элементами разного типа
>>> A = ['а','б','в']
>>> print("A:", A, " Количество элементов:", len(A))
"A: ['а','б','в']  Количество элементов: 3"
```

Встроенная функция len универсальна и может использоваться для вычисления длины любых коллекций. В нашем примере с помощью этой функции возвращается количество элементов в списке.

Оператор сложения + позволяет объединить два списка в один, а не производит поэлементного сложения.

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

К элементам списка можно обращаться по индексу начиная с 0. Поддерживаются также вырезки и сечения то есть извлечения определенного диапазона элементов, для чего используется синтаксис start:stop:step. Следует иметь ввиду, что правая граница никогда в результат не включается.

```
>>> b[0:3]
[5, 4, 3]
>>> b[:3]
[5, 4, 3]
```

```
>>> b[0:4:2]
[5, 3]
>> b[::2] # каждый второй
[5, 3, 1]
```

Список как таковой имеет одномерную структуру, однако можно вкладывать списки друг в друга. Полноценные многомерные массивы доступны при использовании библиотеки NumPy.

```
l = [[11, 12], [21, 22]]
l[0][1] # 12
```

1.4.2. Методы списков

Список является объектом, поэтому у него есть различные методы. Они также доступны с помощью оператора точка `.`. Метод `append` добавляет элемент в конец списка, метод `pop()` удаляет и возвращает последний элемент списка и метод `sort`, который сортирует список по возрастанию или убыванию (если задан параметр `reverse`).

```
>>> l = [1, 2, -1, 4, 5, 6]
>>> l.append(7)
[1, 2, -1, 4, 5, 6, 7]
>>> l.pop()
[1, 2, -1, 4, 5, 6]
>>> l.sort(); l
[-1, 1, 2, 4, 5, 6]
```

Все доступные для использования методы можно посмотреть в документации языка или же — при использование оболочки Jupyter Notebook — нажав клавишу Tab после поставленной точки. Справку по конкретному методу можно получить нажав Shift-Tab после имени метода.

1.4.3. Конструирование списков с помощью списковых сборок

Python позволяет конструировать списки гибким способом, называемым *списковым включением* или *списковой сборкой* (list comprehension).

Начнем рассмотрение с примера, в котором сгенерируем список, содержащий последовательность целых чисел от 1 до 10. Сделаем вначале это с помощью цикла `for`. Создаем пустой список, а затем в цикле заполняем его значениями.

```
l = []
for i in range(1, 11, 1):
    l.append(i)
print(l)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

То же самое можно сделать короче, если воспользоваться специальным синтаксисом списковой сборки. Фактически мы записываем цикл внутри квадратных скобок.

```
l = [i for i in range(1, 11, 1)]
print(l)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

В нашем простом примере можно сконструировать список намного короче, использовав конструктор `list`

```
l = list(range(1, 11, 1))
print(l)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Усложним наш пример, сформировав список квадратов целых чисел. Воспользуемся сперва циклом

```
l = []
for i in range(1, 11, 1):
    l.append(i**2)
print(l)
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

А теперь сделаем то же самое с помощью списковой сборки

```
l = [i**2 for i in range(1, 11, 1)]
print(l)
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

То же самое можно сделать и с помощью конструктора `list` но на этот раз запись не будет короче

```
l = list(i**2 for i in range(1, 11, 1))
print(l)
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Продолжим усложнять наш пример, вычисляя квадраты только четных чисел от 1 до 10. Для этого придется использовать условный оператор `if` для выбора только тех элементов которые делятся на 2 без остатка.

```
l = []
for i in range(1, 11, 1):
    # оператор % позволяет найти остаток от деления
    if i % 2 == 0:
        l.append(i**2)
print(l)
>>> [4, 16, 36, 64, 100]
```

То же самое можно сделать с помощью списковой сборки заметно короче

```
l = [i**2 for i in range(1, 11, 1) if i % 2 == 0]
print(l)
```

Сформируем теперь таблицу умножения воспользовавшись вложенными списками. Сперва сделаем это с помощью цикла, а потом с помощь списковой сборки.

```
l = []
for i in range(1, 10, 1):
    M = []
    for j in range(1, 10, 1):
        M.append(i * j)
    l.append(M)
```

```
l = [[i * j for j in range(1, 10, 1)] for i in range(1, 10, 1)]
```

Добавим теперь условие на четность и сформируем таблицу умножения только четных чисел

```
l = []
for i in range(1, 10, 1):
    if i % 2 == 0:
        M = []
        for j in range(1, 10, 1):
            if j % 2 == 0:
                M.append(i * j)
        l.append(M)
```

```
l = [[i * j for j in range(1, 10, 1) if j % 2 == 0] for i in range(1, 10, 1) if
 ↪  i % 2 == 0]
```

1.4.4. Использование функции zip

Параллельный перебор элементов из двух и более списков можно записать короче, если воспользоваться встроенной функцией `zip`, которая последовательно извлекает из переданных ей списков по элементу и формирует их в кортежи.

Пусть у нас есть список координат `X` и `Y` из которых надо сформировать список точек, в котором каждая точка представлена парой `(x, y)` (кортеж). Как и ранее рассмотрим сперва как это сделать в цикле, а затем в списковой сборке.

```
X = [1, 2, 3, 4, 5, 6]
Y = [1, 2, 3, 4, 5, 6]
Ps = []
for x, y in zip(X, Y):
    # Обратите внимание на двойные скобки
    print(x, y)
    Ps.append((x, y))
print(Ps)
>>> [(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)]
```

```
Ps = [P for P in zip(X, Y)]
print(Ps)
```

```
# Иначе
Ps = list(zip(X, Y))
print(Ps)
```

Важно помнить, что функция `zip()` создает именно *итератор* (а не список) кортежей из переданных ей списков, а значит результатом ее работы можно воспользоваться только один раз. Рассмотрим пример

```
z = zip(r, r)
print("Тип объекта, возвращаемого функцией zip():", type(z))
print("Проходим по элементам z первый раз")
for item in z:
    print(item, end=' ')
# Это итератор, поэтому после использования в цикле память освободилась!
# Повторим еще раз и убедимся, что в z теперь пусто!
print("\nПроходим по элементам z второй раз")
for item in z:
    print(item)
```

Чтобы из итератора сделать список можно воспользоваться функцией `list()`

1.5. Кортежи

В Python определен тип данных, называемый *кортежом* (tuple) и представляющий из себя упорядоченный набор фиксированной длины. В отличие от списка в кортеж является иммутабельным (immutable) объектом, что означает невозможность модификации уже созданного объекта. В случае кортежа это выражается в невозможности добавлять новые элементы, заменять старые или удалять уже имеющиеся. Однако возможно соединить два кортежа в один с помощью оператора `+` создав новый кортеж.

```
>>> l = (1, 2, 3, 4)
>>> l[2] # 3
>>> for item in l:
        print(item, end=' ')
1 2 3 4
# l[0] = 1 # недопустимая операция!
```

Для создания кортежа можно также использовать встроенную функцию `tuple` и синтаксис списковых сборок:

```
>>> t = tuple(i**2 for i in range(11))
# (0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

В кортеже удобно хранить данные, которые не предполагается изменять. Например, это могут быть координаты точки или вектора.

С понятием кортежа связанна операция распаковки переменных. Проиллюстрируем ее примером

```
>>> x, y = (1, 2)
# в результате x = 1, y = 2
>>> (x, y) = [1, 2] # круглые скобки не обязательны
# аналогично x = 1, y = 2
>>> x, y, z, = [1, 2, 3, 4, 5, 6]
# x = 1, y = 2, z = 3
```

Обратите внимание на запятую в последнем примере после переменной `z`. Она необходимо, если количество элементов в наборе справа больше числа переменных слева.

Кортежи встречаются в языке Python повсеместно. Мы уже сталкивались с кортежами и распаковкой переменных, когда использовали функцию `zip` для одновременного итерирования по двум и более массивам. Кроме того, они часто применяются в функциях в случае, если необходимо вернуть несколько значений.

1.6. Словари

Удобный тип данных для задания составных структур — словарь (ассоциативный массив, отображение (map)). Представляет собой расширенный вариант массива, где элементы индексируются не просто числами от 0 до N, а ключами общего вида, которые могут иметь любой базовый тип, однако чаще всего используют строки для придания ключам осмысленных названий.

1.6.1. Создание словаря. Методы словаря

Создать пустой словарь можно несколькими способами:

```
# Пустой словарь
D = {}
print(type(D))
# или используем конструктор
D = dict()
print(type(D))
```

Для задания содержимого словаря используется синтаксис `key : value`

```
D = {
    'Ключ 1': 'Значение 1',
    'Ключ 2': 'Значение 2',
    'Ключ 3': 'Значение 3'
}
```

Можно также использовать конструктор `dict`

```
keys = ['раз', 'два', 'три']
values = [1, 2, 3]
D2 = dict(zip(keys, values))
```

Доступ к элементам словаря осуществляется путем указания ключа

```
D['Ключ 1']
```

Попытка обратиться к несуществующему ключу не методом `get` вызовет ошибку.

Для добавления новых элементов достаточно указать желаемый ключ и присвоить ему соответствующее значение.

```
D['Ключ 4'] = [1, 2, 3, 4]
print(D)
# {'Ключ 1': 'Значение 1', 'Ключ 2': 'Значение 2', 'Ключ 3': 'Значение 3',
 ↪  'Ключ 4': [1, 2, 3, 4]}
```

Операция `+` не работает со словарями, но для объединения двух словарей можно использовать метод `update`

```
new_D = {
    'Ключ 05': ('a', 'b'),
    'Ключ 1': 'абвгд'
}
D.update(new_D)
print(D)
# {'Ключ 1': 'абвгд', 'Ключ 2': 'Значение 2', 'Ключ 3': 'Значение 3', 'Ключ 4':
 ↪  [1, 2, 3, 4], 'Ключ 05': ('a', 'b')}
```

Важно помнить, что сохранение порядка ключей в словаре не гарантируется. В нашем примере порядок сохраняется, но при большем количестве элементов он может нарушится. Если порядок ключей важен, то следует воспользоваться типом данных `OrderedDict` из модуля `collections`

1.6.2. Перебор значений и ключей в цикле

Для перебора элементов словаря применяет цикл `for` и несколько методов словаря.

```
# перебор всех ключей
for key in D:
    print(key)

# перебор всех значений
for value in D.values():
    print(value)

# перебор пар (ключ, значение)
for key, value in D.items():
    print("{0} <=> {1}".format(key, value))
```

Для проверки, существует ли тот или иной ключ в словаре, можно использовать метод `get`

```
if D.get('Ключ 4') is None:
    print('Ключа № 4 не существует')
else:
    print('Ключ 4 существует')
```

1.7. Функции

Функции являются базовым средством для разделения кода сложной программы на независимые, логически обоснованные части. Функции в Python являются объектом, поэтому ими можно манипулировать также, как и другими объектами: присваивать переменным, передавать в функции в виде аргумента, помещать в списки в виде элементов и т.д. Перейдем к знакомству синтаксиса объявления функции.

1.7.1. Задание функции

Для задания функций используется инструкция `def`, сразу после которой необходимо указать желаемое имя функции. Рассмотрим ряд примеров, показывающих задание простейших функций.

```
def func_01():
    """Строка документации"""
    print('Hello world!')

def func_02(x):
    """Нахождение квадрата числа"""
    return x**2

def func_03(x, y): return (x**2, y**2)
```

- Первая функция не принимает никаких аргументов и не возвращает никакие значения — она лишь распечатывает сообщение Hello World! в консоль.
- Вторая функция принимает один аргумент, и возвращает значение этого аргумента возведенное в квадрат.
- Третья функция принимает два аргумента и возвращает кортеж из двух квадратов своих аргументов.

Если тело функции состоит из одной строчки, то можно сэкономить место и записать данную единственную строку сразу после двоеточия. Также желательно каждую функцию сопровождать строкой документации, которая поясняет предназначение функции, возможные типы ее аргументов, возвращаемое значение, показывает примеры ее использования и т.д.

Типы аргументов функции не фиксируются и возможно передавать в качестве аргумента любой объект (duck typing). Однако в теле функции можно предусмотреть средства проверки типов аргументов, а также прописать в документации с какими аргументами функция может работать.

```python
def func_03(n):
    if not isinstance(n, int):
        print('Аргументы функции должны быть целым!')
        return None
    else:
        return n * (n - 1)
```

Однако следует стараться сделать функции более стабильными и попытаться сконвертировать аргумент в нужный тип автоматически

```python
def func_03(n):
    return int(n) * (int(n) - 1)
```

1.7.2. Аргументы функции

Аргументам функции можно присваивать значения по умолчанию. При этом такие аргументы становятся опциональными и их можно не указывать при вызове функции.

```
>>> def func_04(x, y=1):
>>>    return (x**2, y**3)
>>> func_04(2)
(4, 1)
>>> func_04(1, 4)
(1, 64)
```

Чтобы не запоминать порядок следования аргументов, можно указывать их вместе с названиями (keywords arguments)

```
func_04(2, y=2), func_04(x=2, y=2)
# ((4, 8), (4, 8))
```

Два специальных оператора * и ** позволяют передавать внутрь функции произвольные список и словарь с аргументами. В данном примере переменная `arguments` это список со значениями, которые при распаковки будут переданы в функцию в качестве первого, второго, третьего и т.д. аргументов. Переменная `keywords` является словарем и при использовании оператора ** в функцию будут переданы аргументы по ключевым словам. Внутри фунции их можно разобрать в цикле и обработать.

```
def func_05(*arguments, **keywords):
    for arg in arguments:
        print(arg)
    for (key, value) in keywords.items():
        print(key, ":", value)
func_05(1, 2, 3, x=4, y=5, z=6, p=7)
```

```
1
2
```

```
3
x : 4
y : 5
z : 6
p : 7
```

Используя оператор * можно распаковать готовый список внутри функции. Например для функции

```python
def func_06(x, y, z):
    return x + y + z
func_06(*[1, 2, 3])
# 6
```

Аналогично, используя оператор ** можно распаковать словарь. Это особенно удобно, если функция принимает множество необязательных аргументов, которые можно заранее организовать в виде словаря, а потом передать в функцию для обработки.

```python
def func_07(n=1, string='слово'):
    return n * string
args = {'n': 5, 'string': 'текст '}
func_07(**args)
# 'текст текст текст текст текст '
```

1.8. Вопросы для самопроверки

1. Кратко опишите основные особенности языка `Python`.
2. Что такое интерпретатор? Какие интерпретаторы языка `Python` вы знаете? Чем они отличаются друг от друга?
3. Назовите основные отличия языка с динамическим от языка со статическим типизированием. К какому из этих двух видов относится `Python`?
4. Какие основные типы данных языка `Python` вы знаете?
5. В чем разница между списком (`list`) и кортежом (`tuple`) в языке `Python`?
6. Какие инструкции управления потоком выполнения есть в языке `Python`?
7. Что такое списковые сборки и для чего они предназначены?
8. Для чего нужны генераторы? Не дублируют ли они функционал списков?
9. Можно ли обратиться к символу строки по индексу? Можно ли заменить произвольный символ у уже существующей строки?
10. Можно ли обратиться к элементу кортежа по индексу? Можно ли заменить произвольный элемент кортежа?

11. Могут ли списки `Python` хранить данные привольного типа?
12. Какие методами обладает список в языке `Python`?
13. Какую инструкцию следует использовать для сравнения двух переменных булева типа?
14. Какую инструкцию следует использовать для проверки вхождения подстроки в строку?
15. Каким образом можно последовательно перебрать все элементы списка? Можно ли похожим образом поступить с кортежем? Со строкой?
16. Какие функции создают список, кортеж, словарь?
17. Какие методы есть у строки?
18. Что такое словарь в языке `Python`? Какие у него есть аналоги в других языках программирования?
19. Как упорядочены элементы словаря?
20. Как можно последовательно перебрать все элементы словаря не зная его ключей?
21. Как объявляется функция в языке `Python`?
22. Как задать обязательные аргументы функции? Необязательные аргументы?
23. Для чего служат синтаксические конструкции `*args` и `**kwargs`? Приведите пример.
24. Можно ли передавать в функцию другую функцию?
25. Можно ли возвращать из функции другую функцию? Проверьте на примере.
26. Какая специфика есть у списков при передаче их внутрь функций?
27. Используя документацию языка `Python` ответьте на следующие вопросы.
    - Для чего нужны функции `map`, `filter` и `reduce`?
    - Для чего можно использовать встроенные функции `any` и `all`?
    - Как осуществить вывод в консоль без перевода на новую строку в конце вывода?

Глава 2

Библиотека NumPy

2.1. Общая характеристика библиотеки NumPy

Библиотека NumPy (Numerical Python) привносит в Python новый тип данных, который предназначен для хранения и работы с массивами числовых данных, а также реализует большое количество разнообразных функций для манипуляции, генерации, фильтрации, сортировки массивов. Реализованы также разнообразные математические, в том числе используются функции-обертки процедур библиотеки LAPACK, позволяющие решать задачи линейной алгебры.

В случае, если вы используете WinPython или Anaconda Python, то библиотека NumPy уже установлена. В случае Miniconda ее надо установить выполнив команду

```
conda install numpy
```

после чего можно будет импортировать NumPy. Подключение любых библиотек осуществляется командой import. Инструкция as позволяет переименовать импортируемый модуль, чтобы название было короче и им было удобнее пользоваться при наборе кода. Для библиотеки NumPy стандартом стало двухбуквенное сокращение np.

```
import numpy as np
```

После импорта, все функции и подмодули библиотеки NumPy станут доступны посредством оператора точка '.'. Подобная изоляция *пространства имен* разных модулей позволяет избежать конфликта имен и служит инструментом структурирования функционала библиотеки, позволяя распределить функции по тематическим подмодулям.

Массивы NumPy в первом приближении похожи на встроенный в Python тип данных list, но работают гораздо быстрее, так как каждый массив хранят данные только одного типа и их размер задается при инициализации. Это позволяет сократить накладные расходы памяти и уменьшить время доступа. Большинство функций и структур данных библиотеки NumPy реализованы на языках C/C++ и Fortran. Также активно используется векторизация безиндексных операций над массивами, позволяющая задействовать все ядра многоядерных процессоров и распределить работу между ними.

В данной главе дается краткое введение в основные возможности NumPy версии не ниже 1.13.

2.2. Создание NumPy массива. Основные операции

2.2.1. Создание массива

Основной функцией для создания массива служит функция `np.array()`. Она принимает перечисляемые типы данных (`list`, `tuple`, генераторы и т.д.) и создает на их основе `numpy` массив. Переданная структура должна состоять из элементов одного типа. В случае числовых данных допускается комбинация целых, рациональных, вещественных и комплексных числе, однако при создании массива они будут приведены к наиболее общему типу. Рассмотрим ряд примеров.

```
a = np.array([1, 2, 3, 4, 5])
print(a, 'тип элементов', a.dtype)
# если хотя бы один элемент вещественный, то весь массив будет иметь
  ↪ вещественный тип
a = np.array([1, 2.0, 3, 4, 5])
print(a, 'тип элементов', a.dtype)
# то же самое справедливо для комплексного типа
a = np.array([1, 2.0, 3 + 1j, 4, 5])
print(a, 'тип элементов', a.dtype)
# можно передавать строки, тогда результатом будет символьный массив
a = np.array([1, 2.0, 3, '4', 5])
print(a, 'тип элементов', a.dtype)
a = np.array([1, 2.0, 3 + 1j, 'a', 5])
print(a, 'тип элементов', a.dtype)
# Программа даст следующий вывод
# [1 2 3 4 5] тип элементов int64
# [ 1.  2.  3.  4.  5.] тип элементов float64
# [ 1.+0.j  2.+0.j  3.+1.j  4.+0.j  5.+0.j] тип элементов complex128
# ['1' '2.0' '3' '4' '5'] тип элементов <U32
# ['1' '2.0' '(3+1j)' 'a' '5'] тип элементов <U64
```

Тип передаваемых элементов можно указать явным образом с помощью аргумента `dtype`

```
a = np.array([1, 2, 3, 4, 5], dtype=np.float64)
print(a, 'тип элементов', a.dtype)
# [ 1.  2.  3.  4.  5.] тип элементов float64
```

`Numpy` поддерживает стандартные типы данных `Python` и плюс к этому определяет более специфические численные типы, вроде беззнаковых целых (`uint8`, `uint16`, `uint32` и `uint64`), действительных чисел (`float16`, `float32` и `float64`), комплексных чисел (`complex64`, `complex128`).

Для создания многомерных массивов можно использовать вложенные списки и списковые сборки

```
a = np.array([[i+j for i in range(5)] for j in range(5)])
print(a)
#[[0 1 2 3 4]
```

```
# [1 2 3 4 5]
# [2 3 4 5 6]
# [3 4 5 6 7]
# [4 5 6 7 8]
```

Также имеется ряд функций, предназначенных для создания массивов из нулей, единиц, одинаковых элементов или просто для выделения памяти.

```
# Массив, состоящий целиком из нулей
a = np.zeros(5, dtype=int)
b = np.zeros(shape=(5, 2), dtype=int)
# Массив, состоящий целиком из единиц
a = np.ones(shape=(2, 2))
# Единичная матрица 5x5
E = np.eye(5)
```

Функция np.arange(start, stop, step) создает массив из элементов арифметической прогрессии начиная от start и заканчивая stop (но не включая его) с шагом step. Последовательность может состоять из вещественных чисел.

```
a = np.arange(1, 6, 2)
b = np.arange(1.0, 6.0, 2.0)
a, a.dtype, b, b.dtype
```

Функция `np.linspace(a, b, N)` разбивает отрезок (a, b) на N частей. Эту функцию удобно применять для задания области определения кривых.

```
l = np.linspace(0, 1, 20)
```

Наконец, если необходимо лишь выделить память под данные определенного типа, то можно воспользоваться функцией `np.empty`. Значениями будут произвольные, случайно оказавшиеся в соответствующих ячейках памяти данные.

```
a = np.empty(shape=(2, 2), dtype=np.float64)
```

2.2.2. Характеристики массива и доступ к элементам

У каждого `numpy`-массива имеется ряд характеристик, которые хранятся в виде атрибутов.

- Число измерений (целое число, атрибут `ndim`).
- Форма или иначе число элементов по каждому измерению (кортеж из целых чисел, атрибут `shape`).
- Общий размер массива, равный суммарному количеству элементов, содержащихся в массиве (целое число, атрибут `size`).
- Тип данных массива (атрибут `dtype`).

```
a = np.empty(shape=1)
b = np.empty(shape=(2, 3))
c = np.empty(shape=(2, 3, 4), dtype=np.int64)
print('a: ndim={0}, shape={1}, size={2}, dtype={3}'.format(a.ndim, a.shape,
 ↪  a.size, a.dtype))
print('b: ndim={0}, shape={1}, size={2}, dtype={3}'.format(b.ndim, b.shape,
 ↪  b.size, b.dtype))
print('c: ndim={0}, shape={1}, size={2}, dtype={3}'.format(c.ndim, c.shape,
 ↪  c.size, c.dtype))
a: ndim=1, shape=(1,), size=1, dtype=float64
b: ndim=2, shape=(2, 3), size=6, dtype=float64
c: ndim=3, shape=(2, 3, 4), size=24, dtype=int64
```

Стандартным способом доступа к элементам массива является доступ по индексам. Здесь синтаксис сходен со стандартным синтаксисом языка Python и с другими языками, где индексация начинается с нуля. Кроме доступа к конкретным элементам, можно использовать синтаксис срезов, получая доступ сразу к набору элементов с заданным шагом. Точно также, как и в случае списков, последний элемент диапазона среза не входит в результат среза.

```
a = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
# Вся вторая строка
print(a[1, :])
# Весь первый столбец
print(a[:, 0])
# [21 22 23]
# [11 21 31]
```

Перебрать все элементы можно с помощью цикла `for`. Обход при этом будет производиться по строкам.

```
for row in a:
    print('Строка', row, end=' и ее элементы: ')
    for item in row:
        print(item, end=' ')
    print()
# Строка [11 12 13] и ее элементы: 11 12 13
# Строка [21 22 23] и ее элементы: 21 22 23
# Строка [31 32 33] и ее элементы: 31 32 33
```

Функция `np.ndenumerate` является эквивалентом функции `enumerate` и служит для последовательного перебора всех элементов массива, возвращая кортеж `(index, item)`, где `index` в свою очередь является кортежем индексов элемента `item`. Рассмотрим пример.

```
for ((i, j), item) in np.ndenumerate(a):
    print('a[{0}, {1}] = {2}'.format(i, j, item))
# a[0, 0] = 11
# a[0, 1] = 12
```

```
# a[0, 2] = 13
# a[1, 0] = 21
# a[1, 1] = 22
# a[1, 2] = 23
# a[2, 0] = 31
# a[2, 1] = 32
# a[2, 2] = 33
```

Обратите внимание, что и здесь обход производится по строкам. Это связанно со способом хранения элементов массива в ячейках памяти. Так как структуры данных `NumPy` реализованы на языке `C`, то элементы многомерных массивов располагаются в памяти по строкам, поэтому перебор по строкам — это наиболее быстрый способ последовательного их перебора.

Кратко упомним основные функции, которые служат для изменения формы массивов.

- Функция `np.reshape` позволяет переформатировать массив.
- Функции `np.concatenate`, `np.vstack`, `np.hstack` позволяют разными способами слить несколько массивов в один.
- Функции `np.split`, `np.split`, `np.hsplit`, наоборот, разбивают массив на части.

2.3. Алгебраические действия и математические функции

Стандартные циклы `Python` весьма медленны. Это обусловлено динамической природой языка, так как при каждой операции над элементами списка интерпретатор должен проверять тип элемента, который не известен заранее. Библиотека `NumPy` устраняет этот недостаток с помощью *векторизации* операций. Векторизованные операции в библиотеке `NumPy` реализованы посредством *универсальных функций* (ufuncs), главная задача которых состоит в быстром выполнении повторяющихся операций над значениями из массивов библиотеки NumPy.

При использовании больших массивов, следует избегать обращения к элементам по индексам и стараться работать с массивами в безиндексной форме. Например, если необходимо сложить, умножить, вычесть, разделить или возвести в степень все элементы массива, то можно воспользоваться обычными операторами `+`, `*`, `-`, `/` и `**` как показано в примере.

```
x = np.random.random(size=10**6)
y = np.random.random(size=10**6)
%timeit x + y
%timeit x * y
%timeit x - y
%timeit x / y
%timeit x**2
```

```
2.25 ms ± 16.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
2.24 ms ± 8.93 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
2.25 ms ± 9.63 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
4.27 ms ± 2.56 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
1.52 ms ± 4.75 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Если же теперь проделать все те же действия с помощью обычного цикла, обращаясь к элементам массива по индексу, то суммирование будет произведено более чем в 100 раз медленнее.

```
%%timeit
for i in range(10**6):
    x[i] = x[i] + y[i]
```

```
565 ms ± 2.73 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Кроме арифметических операторов, определены универсальные функции для стандартных математических действий, вроде нахождения абсолютного значения всех элементов, среднего, максимального и минимального значений. Также векторизированы элементарные математически функции (тригонометрические, гиперболические, экспонента, логарифмы, квадратный корень). Рассмотрим примеры их использования и оценим ускорение операций, которого они позволяют достичь.

```
%timeit [math.sin(item) for item in x]
%timeit np.sin(x)
```

```
327 ms ± 290 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
45.6 ms ± 43.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit [math.sqrt(item) for item in x]
%timeit np.sqrt(x)
```

```
278 ms ± 280 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
4.26 ms ± 406 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Как видно из результатов, векторизированные функции обеспечивают ускорение на порядок, а в ряде случаев и на два порядка. Однако встроенные функции выигрывают в случае операции над скалярными данными.

```
%timeit math.sin(2.0)
%timeit np.sin(2.0)
```

```
171 ns ± 0.719 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
1.15 µs ± 0.977 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%timeit math.sqrt(2.0)
%timeit np.sqrt(2.0)
```

```
149 ns ± 0.684 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
1.14 µs ± 3.53 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Некоторые из векторизированных функций могут задействовать несколько ядер процессора. Такова, например, функция dot, вычисляющая скалярное произведение двух массивов.

```
%timeit np.dot(x, y)
```

```
956 µs ± 15 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

2.4. Анализ массивов NumPy

2.4.1. Статистические характеристики данных

В библиотеке `NumPy` определены функции для вычисления статистических характеристик данных, содержащихся в массиве. Все эти функции могут применяться как ко всем элементам, так и к элементам лишь вдоль указанной размерности. Перечислим важнейшие из этих функций.

- `np.max`, `np.min` — максимум и минимум.
- `np.sum`, `np.cumsum` — сумма и накопленная (кумулятивная) сумма.
- `np.prod`, `np.cumprod` — произведение или накопленное (кумулятивное) произведение.
- `np.mean`, `np.var` — среднее и вариация (эмпирическая дисперсия).
- `np.percentile` — квантили.

Рассмотрим примеры применения этих функций.

```
a = np.arange(1, 6, 1)
res = """a = {},
max(a) = {}, min(a) = {},
sum(a) = {}, cumsum(a) = {},
prod(a) = {}, cumprod(a) = {},
mean(a) = {}, var(a) = {},
percentile(a, q=50) = {} # медиана
""".format(a, np.max(a), np.min(a), np.sum(a), np.cumsum(a), np.prod(a),
           np.cumprod(a),np.mean(a), np.var(a), np.percentile(a, q=50))
print(res)
```

```
a = [1 2 3 4 5],
max(a) = 5, min(a) = 1,
sum(a) = 15, cumsum(a) = [ 1  3  6 10 15],
prod(a) = 120, cumprod(a) = [  1   2   6  24 120],
mean(a) = 3.0, var(a) = 2.0,
percentile(a, q=50) = 3.0 # медиана
```

```
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# суммирование по первому и второму измерениям, а также по всем элементам
b.sum(axis=0), b.sum(axis=1), b.sum()
```

```
(array([12, 15, 18]), array([ 6, 15, 24]), 45)
```

2.4.2. Логические операции над массивами

Операции сравнения (<, > и т.д.) реализованы в библиотеке `NumPy` как поэлементные векторизованные функции. Пользоваться ими можно также как и арифметическими операторами. В качестве результата будут возвращены *логические маски*. Рассмотрим примеры.

```
a = np.array([1, -1, 2, 0, 3, -2, 1])
print('a > 0', a > 0)
print('a < 0', a < 0)
print('a >= 0', a >= 0)
print('a <= 0', a <= 0)
print('a == 0', a == 0)
```

```
a > 0 [ True False  True False  True False  True]
a < 0 [False  True False False False  True False]
a >= 0 [ True False  True  True  True False  True]
a <= 0 [False  True False  True False  True False]
a == 0 [False False False  True False False False]
```

Если передать результаты операции логического сравнения массиву в качестве индексов, то будут отобраны только те элементы, которые удовлетворяют переданному условию. Рассмотрим пример.

```
print('Элементы a > 0:', a[a > 0])
print('Элементы a < 0:', a[a < 0])
print('Элементы a >= 0:', a[a >= 0])
print('Элементы a <= 0:', a[a <= 0])
print('Элементы a == 0:', a[a == 0])
```

```
Элементы a > 0: [1 2 3 1]
Элементы a < 0: [-1 -2]
Элементы a >= 0: [1 2 0 3 1]
Элементы a <= 0: [-1  0 -2]
Элементы a == 0: [0]
```

Для создания более сложных логических условий можно использовать специальные функции `np.logical_and`, `logical_or`, `logical_not` и `logical_xor`. Они служат заменителями стандартных операторов `and`, `or`, `not` и `xor`. Гораздо удобнее применять побитовые логические операции `&` и `|`. В случае логических логических масок побитовые операторы полностью заменяют обычные логические операции.

```
# все элементы из отрезка [0, 1]
a[(a >= 0) & (a <= 1)]
```

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

Комбинируя логические операции с универсальными функциями можно отфильтровать данные и сразу же применить к ним желаемые операции.

```
a[a>0].sum() # 7
a[(a >= 0) & (a <= 1)].sum() # 2
```

Функция `np.count_nonzero` позволяет вычислить количество ненулевых элементов в массиве. Так как логическое значение `True` интерпретируется как `1`, а значение `False` как `0` то с помощью `np.count_nonzero`, можно проверять массивы на наличие элементов, удовлетворяющих самым разнообразным условиям.

```
# Есть ли элементы из отрезка [0, 1]
np.count_nonzero((a >= 0) & (a <= 1)) # 3
```

Функция `np.any` принимает в качестве аргумента массив с логическими элементами и возвращает истину в случае если в массиве есть хотя бы один истинный элемент. В свою очередь функция `np.all` возвращает истину, если в массиве все элементы истины. Эти функции могут заменить `count_nonzero` в случае, если число элементов, удовлетворяющих условию нам не интересно.

```
np.any(a > 0), np.all(a > 1)
```

Стоит отметить, что в языке `Python` есть встроенные функции `any`и `all`, которые для не очень больших одномерных массивов работают даже быстрее, чем функции `NumPy`. Проверим это утверждение на примере.

```
for i in range(3,7):
    print("Для массива из {0} элементов:".format(i))
    r = np.random.random(10**i)
    %timeit np.any(r>0.5)
    %timeit any(r>0.5)
    %timeit np.all(r>0.5)
    %timeit all(r>0.5)
# Запустите данный код самостоятельно
# в качестве упражнения
```

Однако если `NumPy` функции можно применять и к многомерным массивам, то встроенные нельзя, поэтому `NumPy` функции являются более универсальными.

2.5. Чтение и запись данных

2.5.1. Чтение данных из файла

Библиотека `NumPy` предоставляет ряд средств, которые позволяют считывать данные из файлов разного формата, а также сохранять массивы в виде файлов для дальнейшего использования. Полный список функций, реализующих считывание и запись файлов, можно найти в разделе Input and output официальной документации по ссылке `https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.io.html`. Мы рассмотрим наиболее часто применяемые из этих функций.

Для преобразования данных, хранящихся в текстовом виде, можно использовать две функции `np.loadtxt` и `np.genfromtxt`. Последняя функция может обработать данные, содержащие ошибки или пропуски. В остальном же эти функции похожи. Рассмотрим примеры их работы. Создадим вначале текстовой файл `data.csv`, содержащий данные числовые следующего вида:

```
t,x,y
0.1,0.1,0.2
0.2,0.15,0.1
0.3,0.2,0.15
0.4,0.25,0.2
```

Файлы такого формата получили название CSV-файлов (comma separated values — значения разделенные запятыми) и широко применяются для хранения табличных данных не требующих форматирования. В качестве разделителя также часто используется точка с запятой. В таком формате удобно хранить результаты не очень объемных вычислений. Рассмотрим, как с помощью `np.loadtxt` можно извлечь данные из CSV-файла.

```
data = np.loadtxt(fname='data.csv', skiprows=1, delimiter=',')
print(data)
[[ 0.1   1.1   3.2 ]
 [ 0.2   1.15  2.1 ]
 [ 0.3   1.2   1.15]
 [ 0.4   1.25  2.2 ]]
```

Параметр `skiprows` равный 1 указывает на то, что при считывании необходимо пропустить первую строку, так как в ней содержатся не числовые данные (названия колонок). Параметр `delimiter` задает разделитель. В нашем случае это запятая, но можно задать любой символ. По умолчанию за разделитель считается пробел. По завершении считывания файла функция возвращает двумерный массив. Однако можно считать каждую колонку в отдельную переменную, воспользовавшись параметром `unpack`.

```
 t, x, y = np.loadtxt(fname='data.csv', unpack=True, skiprows=1, delimiter=',')
print(t, x, y)
[ 0.1  0.2  0.3  0.4] [ 1.1   1.15  1.2   1.25] [ 3.2   2.1   1.15  2.2 ]
```

Есть возможность указать тип считываемых значений явным образом с помощью аргумента `dtype`, причем для разных колонок можно указать разный тип данных. Делается это следующим образом.

```
types = {'names': ('t', 'x', 'y'), 'formats': ('f4', 'i4', 'i4')}
data = np.loadtxt(fname='data.csv', dtype=types, skiprows=1, delimiter=',')
```

При этом создается массив специального смешанного типа, к однотипным элементам которого можно обращается по именам полей, которые мы указали в переменной `types`. В случае однородных данных достаточно передать `dtype` один аргумент.

```
data = np.loadtxt(fname='data.csv', dtype=np.float64, skiprows=1, delimiter=',')
```

В случае, если необходимо работать с объемными таблицами с разнородными данными, то предпочтительнее использовать библиотеку `Pandas`, которая специально предназначена для этого.

2.6. Вопросы для самопроверки

Под термином *массив* в вопросах всегда подразумевается массив библиотеки `NumPy`.

1. Дайте краткую характеристику библиотеке `NumPy`.
2. Какое стандартное сокращение принято использовать при импорте библиотеки `NumPy`?
3. Какие недостатки встроенного в `Python` типа `list` призван устранить массив `NumPy`?
4. Какие типы данных поддерживает `ndarray`?
5. Какие функции для создания массивов вы знайте?
6. В каких случаях стоит предпочесть `ndarray` встроенному списку `Python`?
7. Как создать многомерный массив?
8. Массивы какого вида можно создавать с помощью специальных функций `eye`, `ones`, `zeros` и `empty`?
9. Как явно указать тип элементов массива?
10. На каком языке написаны большинство функций `NumPy`?
11. Что такое означает термин векторизация в контексте библиотеки `NumPy`?
12. Что такое универсальные функции и как их использование может ускорить программу?
13. Какие две функции позволяют создать линейную последовательность элементов?
14. Как получить доступ к элементам массива с помощью индексов? С какого индекса начинается нумерация? Как получить доступ к последнему индексу?
15. Как работает синтаксис срезов?
16. Как можно эффективно сравнить все соответствующие элементы двух массивов?
17. Как перебрать все элементы массива пользуясь циклом, но при этом не использовать индексы?
18. Как работают функции `concatenate`, `vstack`, `hstack`?
19. Как работают функции `split`, `split`, `hsplit`?
20. Какие математические функции определены в `NumPy`? В каких случаях их стоит применять?
21. При каких операциях `NumPy` может использовать несколько ядер процессора?
22. Какие функции позволяют вычислять среднее, сумму, кумулятивную сумму элементов массива? Могут ли они работать с многомерными массивами?

23. Какие статистические функции `NumPy` вы знаете?

24. Как создать логическую маску и как ее использовать для доступа к элементам массива?

25. Какие логические операторы следует использовать для составления сложных выражений при сравнении элементов разных массивов сравнения?

26. Какая функция позволяет вычислить количество ненулевых элементов массива? Как с ее помощью вычислить количество любых других одинаковых элементов?

27. Чем функции `any` и `all` из библиотеки `NumPy` отличаются от одноименных встроенных функций?

28. Какие функции позволяют считать данные из файла? Какого формата данные они могут обрабатывать?

29. Охарактеризуйте формат CSV. Для чего его можно использовать?

30. Могут ли в массивах `NumPy` содержаться разнотипные данные?

31. Как сохранить существующий массив на диск? Какой формат данных при этом можно использовать?

32. Попробуйте сформулировать принципы, следуя которым можно добиться наиболее оптимального в смысле быстродействия, использования библиотеки `NumPy`.

33. Какие функции библиотеки `NumPy` могут пригодится при построении кривых и поверхностей?

34. Пользуясь дополнительной литературой и официальней документацией, ответьте на следующие вопросы.

    - Что такое трансляция (broadcasting) массивов? Какие правила автоматической трансляции использует `NumPy`?
    - Какие функции из библиотеки `NumPy` могут использоваться для сортировки массива?
    - Для чего нужна функция `vectorize`?

Глава 3

Библиотека Matplotlib

В данной главе описывается библиотека Matplotlib. Изложение ведется через примеры и пояснения к ним. Естественно, что в ограниченном объеме учебного пособия невозможно описать все аспекты данной библиотеки, поэтому читателю следует в первую очередь обратиться к официальной документации по ссылке http://matplotlib.org/contents.html. Кроме того, большую помощь может оказать галерея примеров, также доступная на официальном сайте. Также данной библиотеки посвящены отдельные главы в книгах [3, 4].

3.1. Общая характеристика библиотеки Matplotlib

Библиотека Matplotlib является одной из наиболее развитых библиотек для визуализации данных не только в рамках языка Python, но и среди других языков программирования и математических пакетов. Сопоставимым уровнем функционала обладают средства для рисования встроенные в коммерческие программы (MatLab, Mathematica, Maple), библиотека ggplot языка R и утилита gnuplot. Однако Matplotlib отличается рядом существенных преимуществ, таких как независимость от платформы, интеграция с библиотекой NumPy, поддержка формул LaTeX. Существенной слабой стороной Matplotlib является плохая поддержка трехмерных построений. В остальном данная библиотека является одним из лучших бесплатных решений для визуализации данных.

3.2. Настройка среды Jupyter

Для использования библиотеки Matplotlib совместно с оболочкой IPython или Jupyter, необходимо выполнить команду %matplotlib inline, а затем импортировать модуль matplotlib.pyp
дав ему краткое имя plt. Именно данный модуль обеспечивает доступ к основным объектам и функциям библиотеки. Также почти во всех примерах будут использоваться функции из библиотеки NumPy, так что следует импортировать и эту библиотеку тоже.

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import matplotlib as np
```

Начиная с версии 1.5 в `Matplotlib` появилась возможность использовать отдельные файлы с настройками стилей изображений. Эти файлы должны иметь расширение `.mplstyle` и содержать параметры, аналогичные параметрам из общего файла настроек `.matplotlibrc`. С помощью функции `plt.style.use` можно загрузить и активировать нужный файл стилей. Все рисунки для данного пособия были выполнены с использованием следующего стилевого файла

```
# black_white.mplstyle
# Настройка стиля для черно-белых картинок
figure.figsize: 10, 4.0
figure.dpi: 200
# Настройка шрифтов
font.family: serif
font.serif: DejaVu Serif, Times New Roman
font.sans-serif: DejaVu Sans, Arial
font.monospace: DejaVu Sans Mono, Consolas, Courier New
font.size: 11
lines.markersize: 3
# Настройки сетки координат
axes.grid: True
grid.linewidth: 0.5
grid.linestyle: dashed
grid.color: gray
axes.prop_cycle: ((cycler('color', ['k']) * cycler('linestyle', ['solid',
 ↪  'dashed', 'dashdot', 'dotted'])) * cycler('marker', [' ', '.', '^']))
```

3.3. Создание изображения и системы координат

Изначально `matplotlib` разрабатывался с оглядкой на `MatLab`, поэтому в библиотеку встроен интерфейс, во многом повторяющий `MatLab`. Мы, однако, будем пользоваться объектно-ориентированным интерфейсом `matplotlib`, так как он обеспечивает большую гибкость и контроль над настройками графиков.

Построение любого графика начинается с создания изображения (объект класса `figure`) и добавления на это изображение одной или нескольких систем координат (объекты класса `axes`). Для создания изображения используется метод `plt.figure()`, который возвращает объект, представляющий из себя пустое изображение без осей, графиков и надписей. В дальнейшем можно добавлять на этот рисунок необходимые элементы. Метод `figure` может принимать ряд аргументов, полный список который можно посмотреть в справочной строке данного метода (в Jupyter Notebook нажать клавишу таб после имени метода).

Следующим шагом необходимо добавить в объект `figure` оси координат. Стандартный метод `add_subplot()` позволяет задать прямоугольную сетку из субкоординат (субграфиков). Рассмотрим пример.

```
fig01 = plt.figure(num=0)
ax01 = fig01.add_subplot(2, 1, 1)
ax02 = fig01.add_subplot(2, 1, 2)
```

В данном примере изображение (см. рис. 3.1) разбивается на две части по горизонтали. Первый аргумент `add_subplot()` — число строк, второй число столбцов и третий номер субграфика. С каждым субграфиком можно взаимодействовать отдельно (переменные `ax01` и `ax02`), строя на нем кривые, меняя тип координатных линий, добавляя аннотации, легенды и подписи к осям.

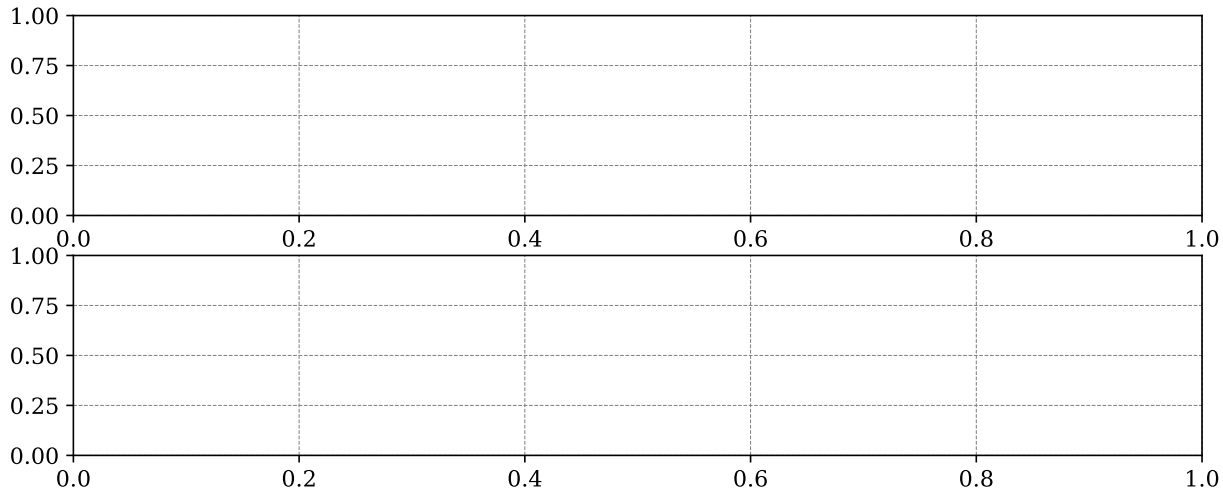

Рис. 3.1: Изображение `fig01`

Добавим к созданным системам координат графики синуса и косинуса (рис. 3.2), для чего сперва рассчитаем соответствующие значения.

```
t = np.linspace(-4*np.pi, 4*np.pi, 400)
x = np.sin(t)
y = np.cos(t)

fig01 = plt.figure(num=0)

ax01 = fig01.add_subplot(2, 1, 1)
ax02 = fig01.add_subplot(2, 1, 2)

ax01.plot(t, x)
ax02.plot(t, y)
```

Если необходимо создать большое число субграфиков в сетке, как на рис. 3.3, то удобно сделать это в цикле или в списковой сборке, как это показано в примере ниже

```
fig02 = plt.figure(num=1)
ax = [fig02.add_subplot(2, 2, i) for i in range(1,5)]

# аналогичный способ с циклом
# ax = []
# for i in range(1,5):
#     ax.append(fig02.add_subplot(2, 2, i))
```

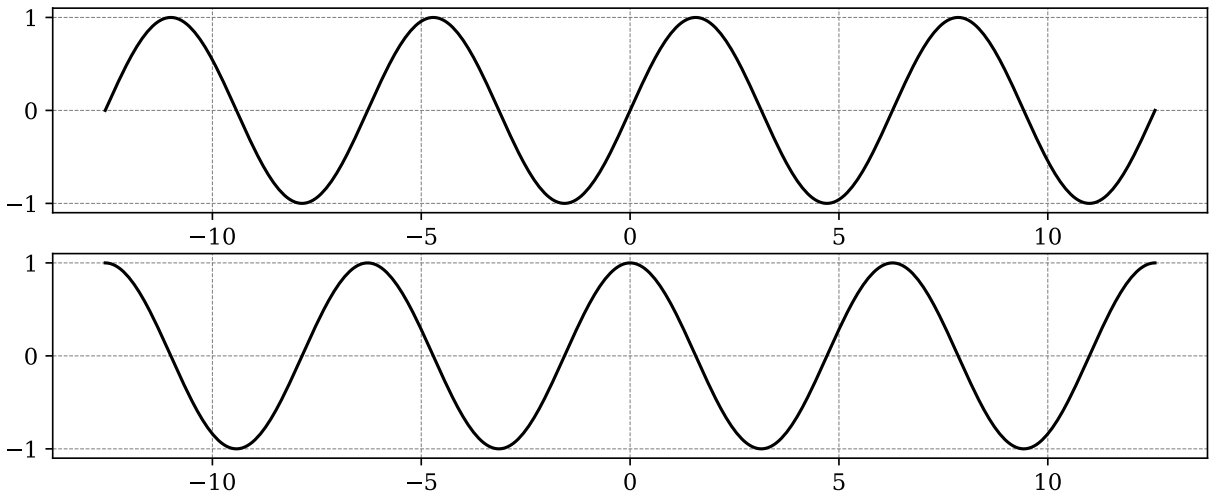

Рис. 3.2: Изображение `fig01` после добавления графиков

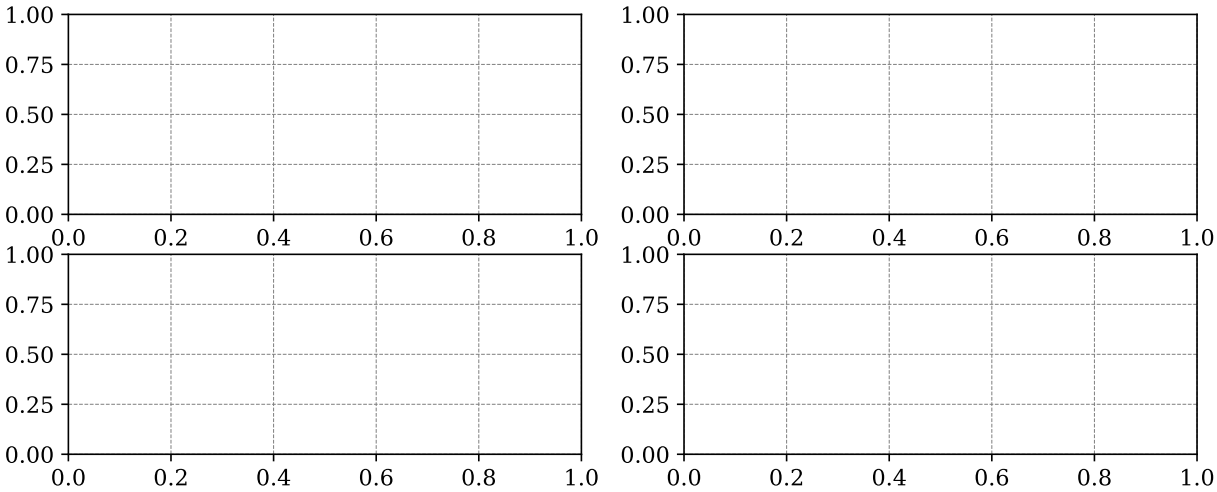

Рис. 3.3: Изображение `fig02`

В примере ниже (рис. 3.4) на каждом субграфике строится функция $y = \sin(ix) + \cos(ix)$, где i — номер субграфика (начиная с 0).

```
fig03 = plt.figure(num=2)
axs = [fig03.add_subplot(3, 3, i) for i in range(1,10)]

for (i, ax) in enumerate(axs):
    y = np.sin(i*x) + np.cos(i*x)
    ax.plot(x, y, linewidth=0.4)

fig03.tight_layout()
```

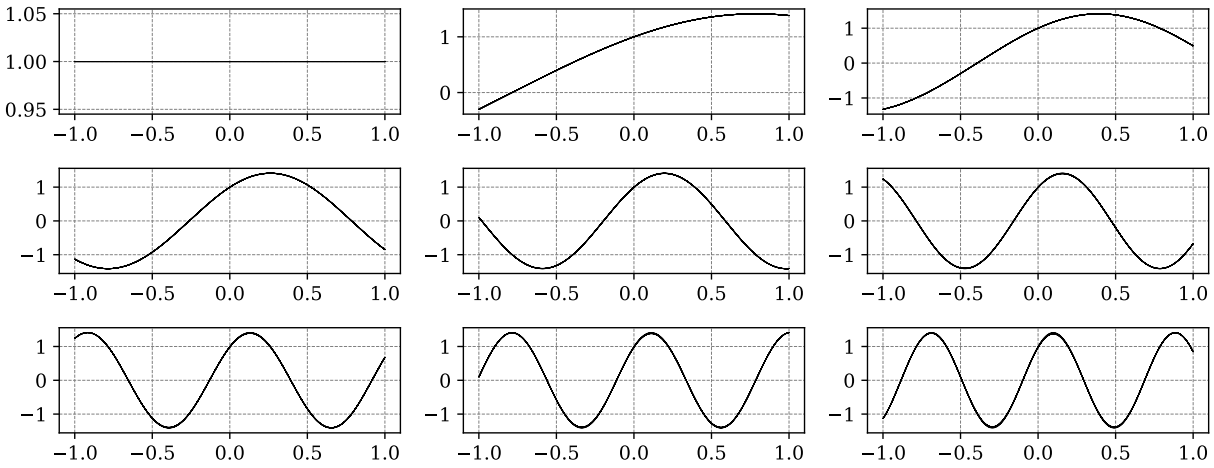

Рис. 3.4: Изображение `fig03`

3.4. Метод `plot`

Метод `plot` служит основным средством для создания одномерных графиков из массива координатных точек. Кроме непосредственно набора координат он может принимать множество параметров для тонкой настройки вида графика. Перечислим наиболее часто используемые из них.

- `color` — цвет; можно задавать любой цвет, если строить несколько графиков, то они автоматически красятся разным цветом.
- `linewidth` — толщина линии, по умолчанию равна 1.
- `linestyle` — тип линии кривой; среди поддерживаемых типов сплошная линия (`solid` или `-`), пунктирная (`dashed` или `--`), точечная (`doted` или `:`) и пунктирно-точечная (`dashdot` или `-.`).
- `label` — легенда графика (то есть краткое пояснение к графику, обычно отображающееся в углу изображения). Следует отметить, что для отображения легенды следует вызвать метод `legend` объекта `axes` иначе легенда не отобразится. Рассмотрим пример использования метода `plot` для отображения двух графиков в одной системе координат в черно-белом формате.

```
t = np.linspace(-4*np.pi, 4*np.pi, 400)
x = np.sin(t) + np.cos(t) + np.cos(2*t)
y = np.cos(2*t) + np.sin(np.cos(t))

fig04 = plt.figure(10)
ax04 = fig04.add_subplot(1, 1, 1)
label_01 = r'$\sin({t}) + \cos({t}) + \cos({2t})$'
label_02 = r'$\cos{(2t)} + \sin{[\cos{(t)}]}$'
ax04.plot(t, y, linewidth=1.0, linestyle='-', marker='None', label=label_01)
```

```
ax04.plot(t, x, linewidth=0.5, linestyle='-', marker='.', markevery=5,
    markersize=4,  label=label_02)

ax04.legend(ncol=2, title='Легенда', framealpha=0.7, fontsize=12)
```

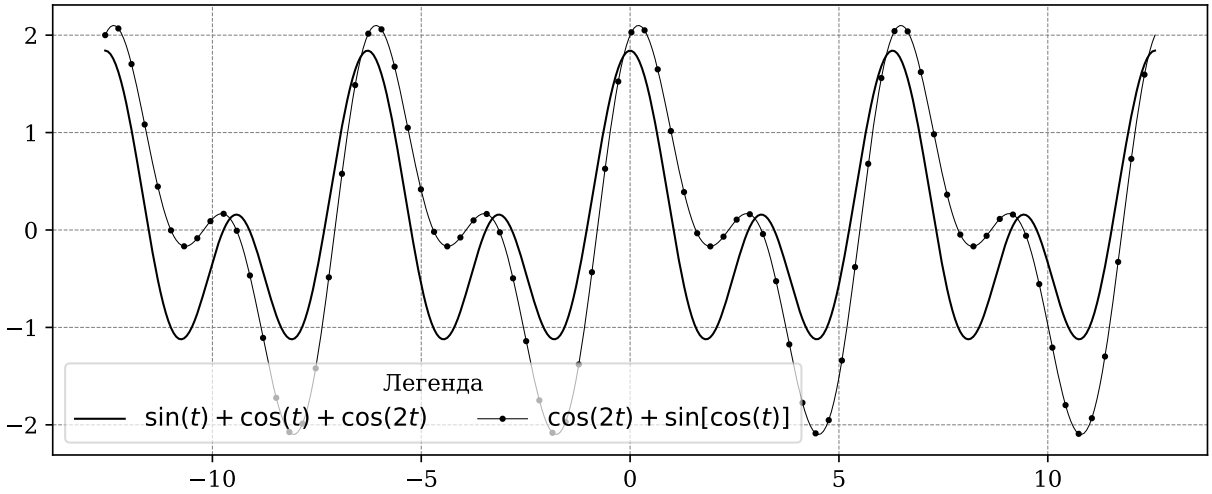

Рис. 3.5: Изображение `fig04`

В данном примере (рис. 3.5) кроме вышеперечисленных опций функции `plot` были использованы опции для настройки маркеров — значков, которыми можно выделить точки графиков. Особенно это полезно на черно-белых изображениях, так как дает возможность различить графики нарисованные одним и тем же цветом. Параметр `marker` устанавливает значек маркера (полный список смотрите в документации к `plot`), параметр `markevery` позволяет регулировать количество маркеров, отображаемых на кривой, `markersize` позволяет установить размер маркера.

В методе `legend` мы использовали параметр `ncol`, который устанавливает количество колонок в легенде, параметр `title` устанавливает заголовок для легенды (по умолчанию заголовок не отображается), параметр `framealpha` регулирует прозрачность рамки легенды, `fontsize` устанавливает размер шрифта. Последним параметром не стоит злоупотреблять, так как гораздо удобнее установить один размер шрифта для всех элементов графика (как это сделать мы рассмотрим далее).

3.5. Настройка осей координат

Продолжим настраивать наш график из прошлого примера. Добавим подписи к осям Ox и Oy с помощью методов `set_xlabel` и `set_ylabel`, заголовок к изображению с помощью метода `set_title`, а также настроим точные границы осей координат с помощью метода `set_xlim`.

```
ax04.set_title('Это заголовок нашей картинки')
ax04.set_xlabel(r'Ось $x$')
```

```
ax04.set_ylabel(r'Ось $y$')

ax04.set_xlim(left=t[0], right=t[-1])
```

Проведем теперь более сложные настройки и заменим отсечки координат по оси `Ox` не просто вещественными числами, а числами кратными π. Для этого сперва создадим два списка. В `xticks` запишем точки -4π, -3π, -2π, $-\pi$, 0, π, 2π, 3π, 4π, а в список `xtickslabels` запишем значения строкового типа, которые предполагается отображать в качестве обозначения отсечек. Можно использовать команды LaTeXдля отображения буквы π или символ из кодировки Юникод π. После того, как списки созданы, их надо передать методам `set_xticks` и `set_xticklabels` которые заменят автоматически созданные отсечки на созданные нами. Результат показан на рис. 3.6

```
xticks = np.linspace(-4*np.pi, 4*np.pi, 9)
xtickslabels = []

for xtick in xticks:
    i = int(xtick/np.pi)
    if i == 1:
        xtickslabels.append(r'$\pi$')
    elif i == -1:
        xtickslabels.append(r'-$\pi$')
    elif i == 0:
        xtickslabels.append(r'0')
    else:
        xtickslabels.append(r'{0}$\pi$'.format(i))

ax04.set_xticks(xticks)
ax04.set_xticklabels(xtickslabels)
fig04
```

Часто бывает необходимо построить два графика в отдельных системах координат, но при этом близко друг к другу с общими отсечками по одной из осей. Рассмотрим следующий пример (рис. 3.7).

```
fig05 = plt.figure(num=4, figsize=(10, 2))

axs05 = [fig05.add_subplot(1, 3, i) for i in range(1,4)]

x = np.linspace(-2,2,100)
funcs = [x, x**2, x**3]

y_min = np.min(funcs)
y_max = np.max(funcs)

for (f, ax) in zip(funcs, axs05):
```

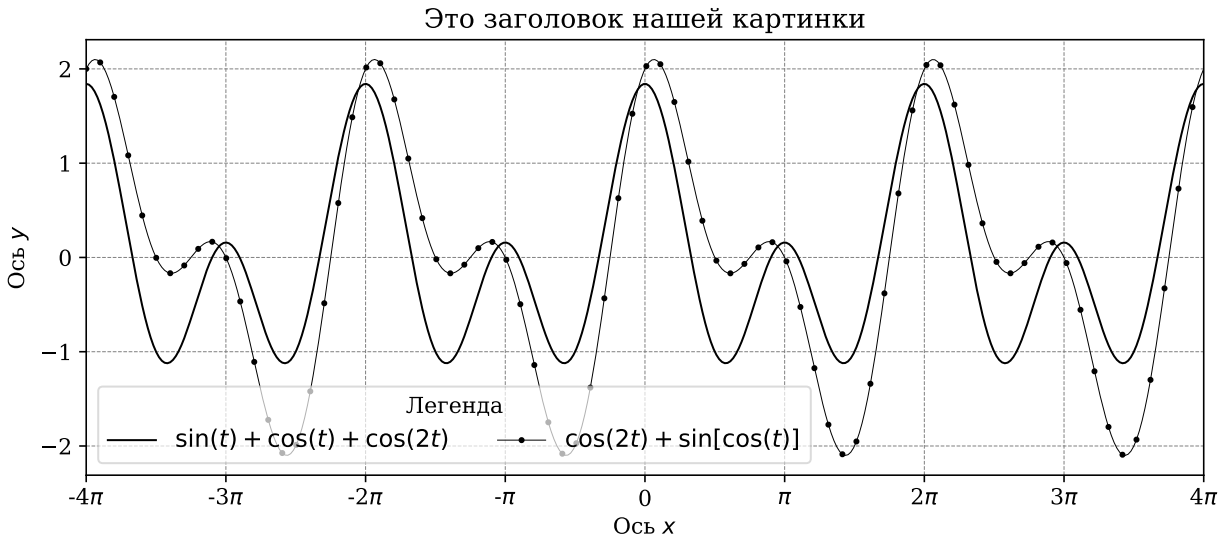

Рис. 3.6: Изображение `fig04` после модификации осей

```
    ax.plot(x, f)
    ax.set_ylim(bottom=y_min, top=y_max)

for ax in axs05[1:]:
    ax.set_yticklabels([])

fig05.subplots_adjust(wspace=0)
```

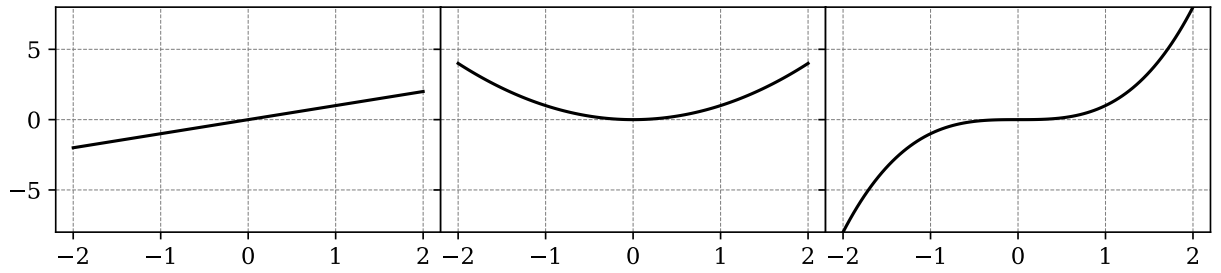

Рис. 3.7: Изображение `fig05`

3.6. Дополнительные способы создания сетки субграфиков

Если необходимо создать большое количество субграфиков, то можно воспользоваться функцией `plt.subplots`. Эта функция создаст целую сетку субграфиков и вернет объект `figure` и массив объектов `axes`. В качестве аргументов `subplots` принимает количество строк и столбцов, а также необязательные логические аргументы `sharex` и `sharey` которые

позволяют установить общую разметку координат. Проиллюстрируем работу функции на примере.

```
rows_number = 3; columns_number = 4
fig06, ax06  = plt.subplots(nrows = rows_number, ncols = columns_number,
 ↪  sharex=True, num=6)

for (i, j), ax in np.ndenumerate(ax06):
    ax.set_title("(стр.: {0}, кол.: {1}).".format(i+1, j+1))

t = np.linspace(-2*np.pi, 2*np.pi, 200)
ticklabels = ['$-2\pi$','$-\pi$','$0$','$\pi$','$2\pi$']

for (i, j), ax in np.ndenumerate(ax06):
    ax.plot(t, np.sin((i+j+1)*t), linewidth=0.5)
    ax.xaxis.set_ticks(np.linspace(-2*np.pi,2*np.pi,5))
    ax.xaxis.set_ticklabels(ticklabels)

fig06.tight_layout()
```

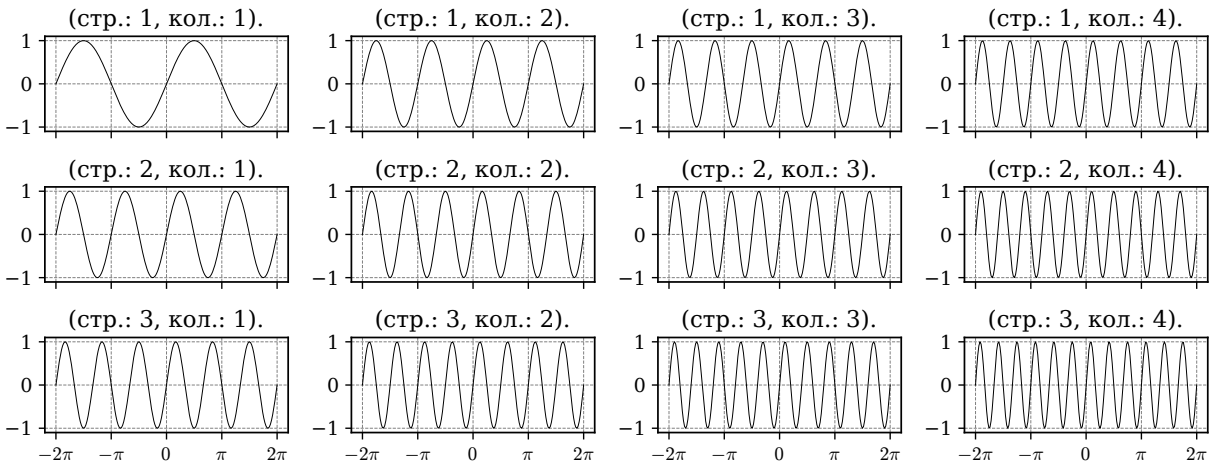

Рис. 3.8: Изображение `fig06`. Иллюстрируется применение функции `subplots`

Для обхода массива `ax06` мы использовали функцию `ndenumerate` библиотеки `NumPy` которая работает аналогично Встроенной функции `enumerate`, но возвращает в случае двумерного массива кортеж индексов `(i, j)`, который мы сразу же распаковываем в соответствующие индексы. Опция `sharex=True` убирает разметку оси Ox всех субграфиков кроме нижнего ряда. Это экономит место и делает изображение более чистым.

Если необходимо задать неоднородную сетку субграфиков, то можно воспользоваться функцией `gridspec.GridSpec` для чего необходимо предварительно импортировать соответствующий подмодуль `import matplotlib.gridspec as gridspec`. Эта функция возвращает массив, с помощью которого можно затем задать местоположение и размеры субграфиков

с используя обычный синтаксис срезов языка Python. Применение `GridSpec` иллюстрирует следующий пример.

```
fig07 = plt.figure(num=7)
gs = gridspec.GridSpec(3, 3, left=0.0, right=0.9, wspace=0.4, hspace = 0.7)
ax07 = []
ax07.append(fig07.add_subplot(gs[0, 0:]))
ax07.append(fig07.add_subplot(gs[1, 0:1]))
ax07.append(fig07.add_subplot(gs[1, 1:]))
ax07.append(fig07.add_subplot(gs[2, 0:2]))
ax07.append(fig07.add_subplot(gs[2, 2:]))

for i, ax in enumerate(ax07):
    ax.set_title("Заголовок {0}".format(i))
    ax.set_ylabel("$Oy$")
```

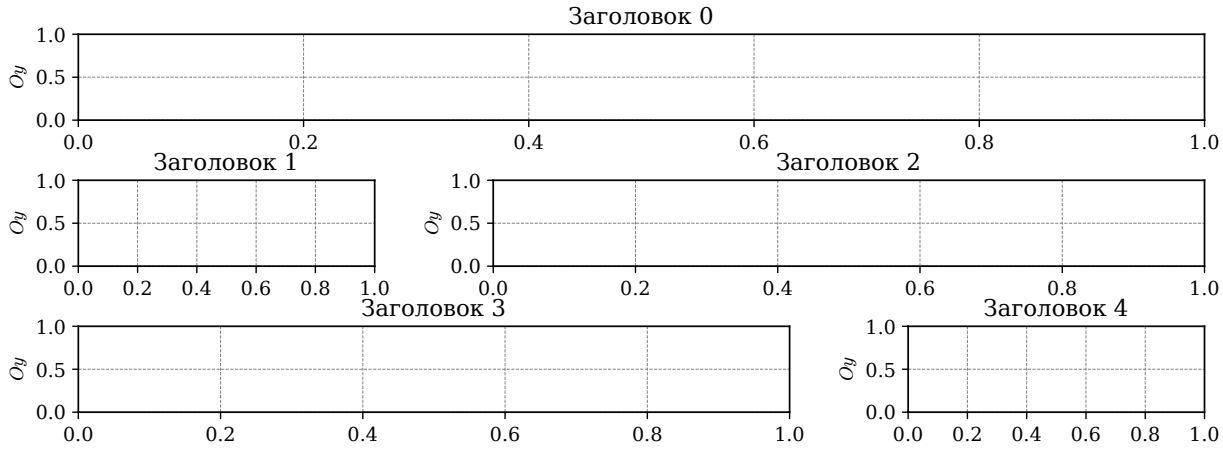

Рис. 3.9: Изображение `fig07`. Иллюстрируется применение `GridSpec`

Наконец максимально гибкое средство настройки сетки субграфиков произвольного вида, это функция `plt.axes`. Данная функция принимает необязательный аргумент, представляющий собой список из четырех чисел в системе координат рисунка. Эти числа означают левый угол, низ, ширину и высоту в системе координат рисунка, отсчет которых начинается с 0 в нижнем левом и заканчивается 1 в верхнем правом углу рисунка. Первая пара из списка фактически представляет собой координаты левого нижнего угла прямоугольника, а вторая права координаты правого верхнего угла.

С помощью метода `axes` можно встраивать малые субграфики прямо в область координат большого графика, что может пригодится для отображения увеличенной окрестности особых точек, например точек пересечения двух кривых, как в примере ниже.

```
fig08 = plt.figure(num=8)
ax08 = [
    fig08.add_axes([0.0, 0.0, 1.0, 1.0]),
```

```
    fig08.add_axes([0.20, 0.6, 0.2, 0.3], xlim=(-1.5, 0), ylim=(-3, 10)),
    fig08.add_axes([0.65, 0.6, 0.2, 0.3], xlim=(1.5, 2.1), ylim=(0, 10))
]
t = np.linspace(-2.0, 2.1, 500)
y = t**4 - t**3 + t**2 - t - 1
z = t**3 - t**2 + 3
for ax in ax08:
    ax.plot(t, y, linewidth=1.0)
    ax.plot(t, z, linewidth=1.0, markevery=10)
    ax.set_xlabel(r'Ось $x$'); ax.set_ylabel(r'Ось $y$')
```

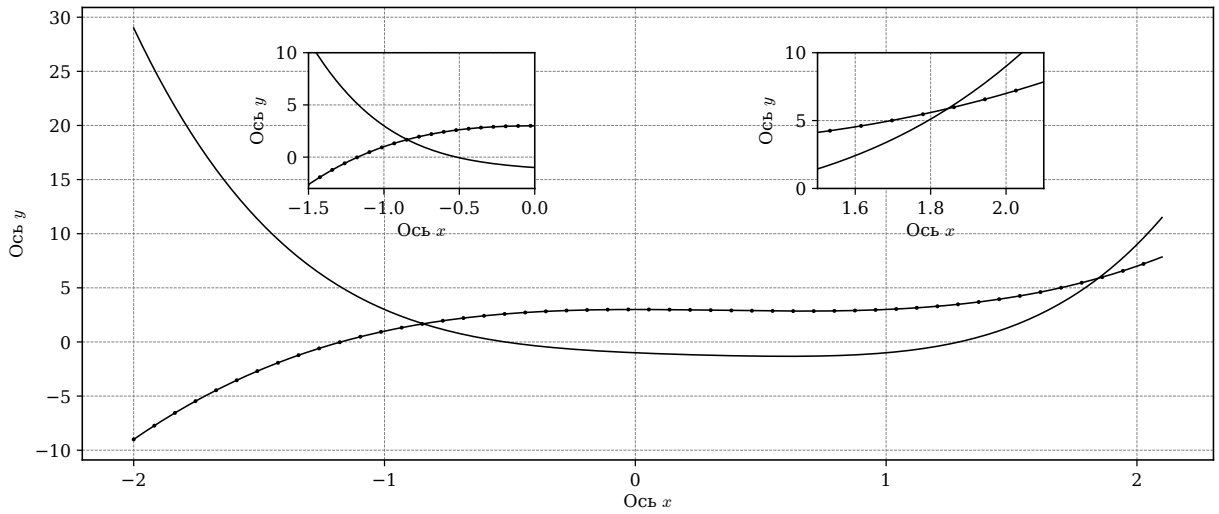

Рис. 3.10: Изображение `fig08`. Иллюстрируется применение `add_axes`

3.7. Манипуляция осями координат. Полярные координаты

Как видно из примеров выше, по умолчанию координатные оси отображаются не по центру, а сбоку. Чаще всего такое расположение осей является оптимальным, так как не мешает восприятию графической информации. Однако есть возможность переместить оси координат в центр графика. Как это сделать показывает следующий пример.

```
fig09 = plt.figure(num=9)
ax09 = fig09.add_subplot(1, 1, 1)

t = np.linspace(-4*np.pi, 4*np.pi, 200)
y = np.sin(t)
ax09.plot(t, y)
# Убираем верхнюю и правую оси координат
```

```
ax09.spines['top'].set_visible(False)
ax09.spines['right'].set_visible(False)

# ‘center’ -> (‘axes’,0.5) ‘zero’ -> (‘data’, 0.0)
# Переносим оставшиеся две оси в центр относительно графика
ax09.spines['bottom'].set_position('zero')
ax09.spines['left'].set_position('zero')

# настраиваем засечки на координатных линиях
ax09.xaxis.set_tick_params(direction='inout', length=10)
ax09.yaxis.set_tick_params(direction='inout', length=10)
```

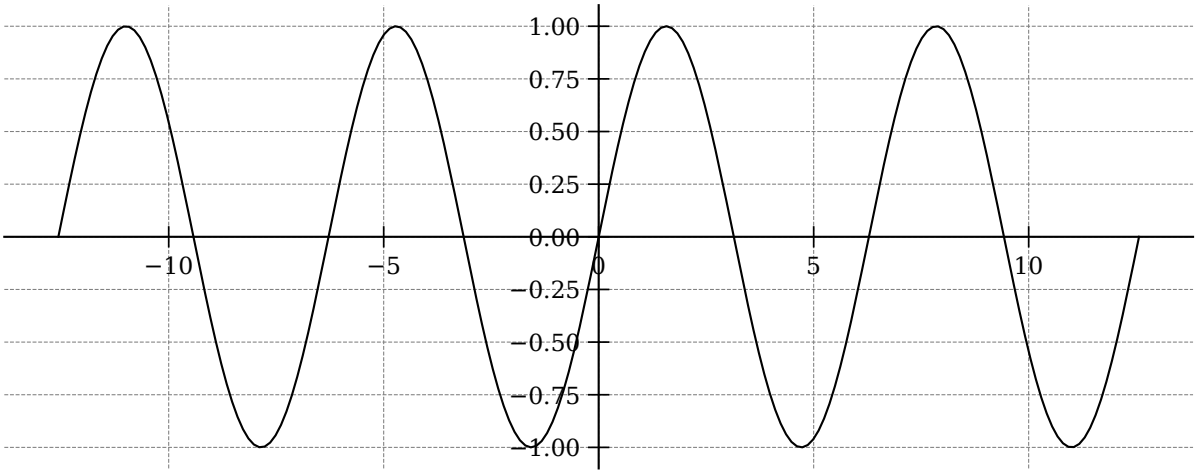

Рис. 3.11: Изображение `fig09`. Иллюстрируется перенос осей координат

Уравнения некоторых кривых сильно упрощаются, если совершить переход в полярную систему координат. В matplotlib есть возможность использовать полярную систему вместо декартовой. Рассмотрим на примере как это сделать.

```
fig10 = plt.figure(num=10, figsize=(3,3))
ax10 = fig10.add_subplot(1, 1, 1, projection='polar')
phi = np.linspace(0.0, 10*np.pi, 1000)
r = 5*np.cos(2.5*phi-1)
ax10.plot(phi, r, linewidth=1.0)
```

В компьютерной геометрии, однако, полярная система координат используется не часто, так как обычно задача заключается в отображении кривых и поверхностей на экране монитора, который имеет прямоугольную форму, поэтому чаще всего используется декартова система координат и параметрические уравнения кривых.

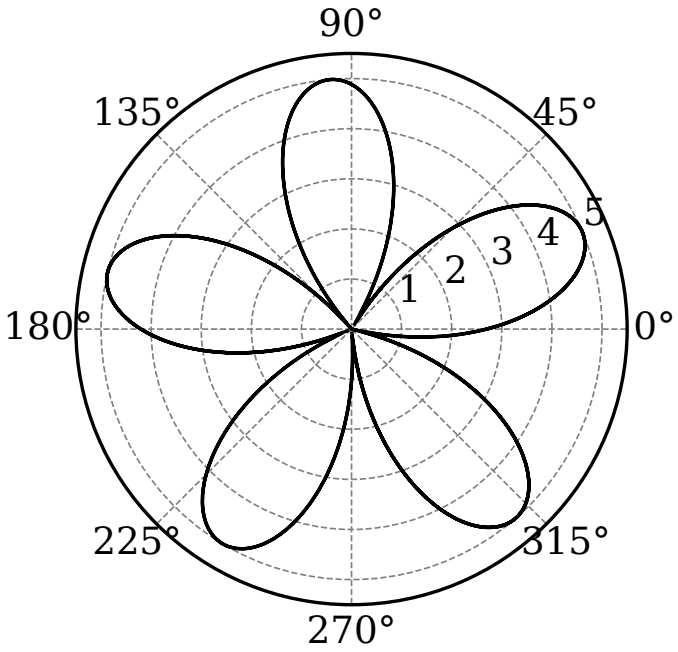

Рис. 3.12: Изображение `fig10`. Иллюстрирует полярную систему координат

3.8. Создание примитивов

Под термином примитив понимаются простейшие геометрические объекты: отрезки, окружности, треугольники и прямоугольники. Для их рисования можно использовать специальные методы, которые позволят, например, добавить на рисунок окружность, указав лишь координаты ее центра и радиус. В `matplotlib` функции для создания примитивов находятся в подмодуле `patches` и `lines`.

Перечислим некоторые из функций подмодуля `patches`.

- `Circle` — создает окружность с центром в точке с координатами `xy` и радиусом `radius`.
- `Ellipse` — создает эллипс с центром с координатами `xy` и полуосями `width` и `height`.
- `Rectangle` — создает прямоугольник, левый нижний край которого имеет координаты `xy`. Высота и ширина прямоугольника регулируется параметрами `width` и `height`, а также его можно вращать вокруг левого нижнего угла задав параметр `angle` (в градусах).
- `Arc` — создает дугу эллипса. Кроме параметров самого эллипса (см. `Ellipse`), принимает параметры `theta1` и `theta2`, которые задают начальный и конечный углы дуги, а также параметр `angle`, который позволяет вращать дугу вокруг центра как единое целое.
- `Wedge` — создает клин (сегмент окружности) с центром в точке `center` и радиусом `r`. Отличие от дуги эллипса `Arc` заключается в прорисовке радиусов. Размер центрального угла задается параметрами `theta1` и `theta2`, также как у дуги эллипса.

- `Polygon` ломанная линия (замкнутая или не замкнутая) произвольного вида, задаваемая массивом координат `xy`. Принимает логический параметр `closed`, который отключает или включает автоматическое замыкание ломанной (соединяет начальную и последнюю точки).

- `Arrow` и `FancyArrow` — создают стрелку с началом в точке с координатами `x`, `y` и проекциями длины `dx` по оси *Ox* и `dy` по оси *Oy*. `FancyArrow` в добавок позволяет более гибко настроить внешний вид стрелки.

- `FancyBboxPatch` — создает прямоугольник со скругленными краями.

Каждая из вышеперечисленных функций принимает в качестве необязательных параметров стандартные аргументы, управляющие внешним видом линии: `linestyle`, `linewidth`, `color` и т.д. Особенно полезен логический аргумент `fill` который включает или отключает заливку фигуры сплошным цветом.

После создания примитива его необходимо добавить на изображения. Для этого используется метод `add_patch` объекта `axes`. Пример ниже показывает использование всех вышеперечисленных примитивов.

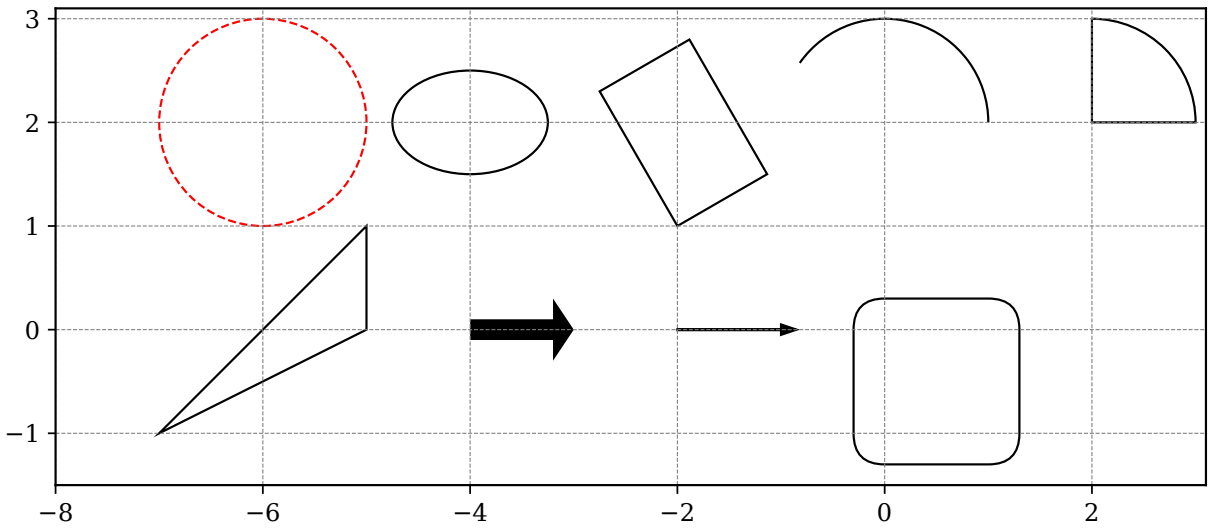

Рис. 3.13: Изображение `fig11`. Иллюстрирует доступные в `matplotlib` примитивы

```
fig11 = plt.figure(num=11)
ax11 = fig11.add_subplot(1, 1, 1)

ax11.set_xlim(left=-8, right=3.1)
ax11.set_ylim(bottom=-1.5, top=3.1)

ax11.set_aspect('equal')
# можно использовать универсальный метод set
# ax11.set(xlim=(-2, 2), ylim=(-2, 2), aspect='equal')

```

```
# Окружность
circle = mpatches.Circle(xy=(-6, 2), radius = 1, color='red', linestyle='--',
 ↪  fill=False)
# Дуга эллипса
elipse = mpatches.Ellipse(xy=(-4, 2), width=1.5, height=1, fill=False)
# Прямоугольник
rectangle = mpatches.Rectangle(xy=(-2, 1), width=1, height=1.5, angle=30,
 ↪  fill=False)
# Арка эллипса
arc = mpatches.Arc(xy=(0, 2), width=2, height=2, angle=0, theta1=0, theta2=145)
# Сегмент окрудности (клин)
wedge = mpatches.Wedge(center=(2, 2), r=1, theta1=0, theta2=90, fill=False)
# Полигон -- ломанная линия (замкнутая или не замкнутая) произвольного вида.
polygon = mpatches.Polygon(xy=np.array([[-7, -1], [-5, 1], [-5, 0]]),
 ↪  closed=True, fill=False)
# Стрелка
arrow = mpatches.Arrow(x=-4, y=0, dx=1, dy=0)
# Стрелка с возможностью настройки внешнего вида
farrow = mpatches.FancyArrow(x=-2, y=0, dx=1, dy=0, width=0.01, head_width=0.1)
# Прямоугольник с закругленными углами
fbox = mpatches.FancyBboxPatch(xy=(0, -1), width=1, height=1.0, fill=False)

ax11.add_patch(circle); ax11.add_patch(elipse); ax11.add_patch(rectangle);
 ↪  ax11.add_patch(arc)
ax11.add_patch(wedge); ax11.add_patch(polygon); ax11.add_patch(arrow);
 ↪  ax11.add_patch(farrow); ax11.add_patch(fbox)
```

В подмодуле `lines` находятся низкоуровневые функции, применяемые для соединения точек ломанными или гладкими кривыми разного порядка. Из всех этих функций, мы рассмотрим лишь одну — `Line2D`. Данная функция принимает два массива координат: по оси Ox (`xdata`) и по оси Oy (`ydata`). Кроме этих параметров, есть возможность очень гибко настроить вид линии. В примере ниже мы добавили параметр `marker`, который добавляет маркеры в каждой из вершин получившейся ломанной.

```
fig12 = plt.figure(num=12, figsize=(10, 1.5))
ax12 = fig12.add_subplot(1, 1, 1)

ax12.set_xlim(left=-1.1, right=1.1)
ax12.set_ylim(bottom=-0.1, top=1.1)

dots = {'xdata': [-1, 0, 1], 'ydata': [0, 1, 0]}
line = mlines.Line2D(**dots, marker='o')

ax12.add_line(line)
```

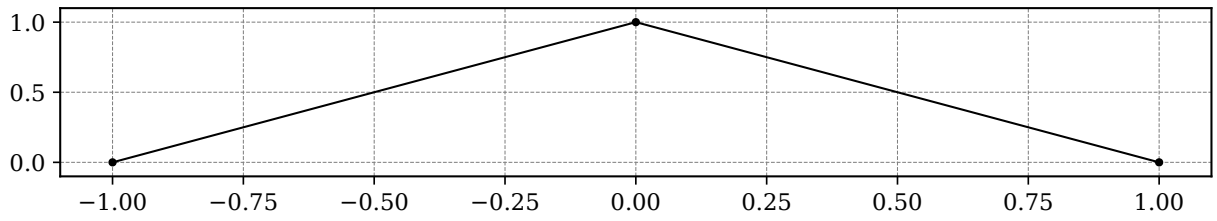

Рис. 3.14: Изображение `fig12`. Иллюстрирует применение `Line2D`

Обратите внимание, что мы сперва создали словарь `dots`, а потом передали его в функцию с помощью оператора `**`. Данный оператор распаковывает словарь и передает в функцию именованные аргументы с теми же именами, что и ключи словаря. Это полезно в случае, если мы вычисляем точки ломанной отдельно, например, считываем из файла или получаем в `JSON`-формате. Естественно, что можно обойтись более стандартным синтаксисом и записать следующий код:

```
xs = [-1, 0, 1]
ys = [0, 1, 0]
line = mlines.Line2D(xdata=xs, ydata=ys, marker='o')
```

3.9. Аннотирование изображений

3.9.1. Текстовые пометки

Библиотека `matplotlib` предоставляет широкие возможности по аннотированию созданных графиков. Основными средствами для этого служат методы `text` и `annotate` класса `axes`. Метод `text` позволяет добавить текст возле любой точки на координатной плоскости. При этом внешний вид текста можно гибко настраивать, в том числе включать LaTeXформулы. Метод `text` требует три обязательных аргумента: `x`, `y` — координаты точки, относительно которой будет отображаться текст и строка `s`, содержащая непосредственно сам текст. Кроме этих аргументов можно передавать большое число дополнительных параметров. Некоторые из них показаны в примере ниже.

```
fig13 = plt.figure(num=13)
ax13 = fig13.add_subplot(1, 1, 1)

ax13.set_xlim(0, 1.5)
ax13.set_ylim(0, 1.5)

points = [(0.25, 0.25), (0.50, 0.50), (0.75, 0.75), (1.0, 1.0), (1.25, 1.25)]

xs = [p for (p, _) in points]
ys = [p for (_, p) in points]

ax13.plot(xs, ys, marker='o', markersize=6, linestyle='None')
```

```
13
14 x, y = points[0]
15 ax13.text(x=x, y=y+0.1, s='Точка №1', color='gray')
16 x, y = points[1]
17 ax13.text(x=x, y=y+0.1, s='Точка №2', rotation_mode='anchor', rotation=90)
18 x, y = points[2]
19 ax13.text(x=x, y=y+0.1, s='Точка №3', horizontalalignment='center',
    ↪ fontstyle='italic', fontsize=14)
20 x, y = points[3]
21 ax13.text(x=x, y=y+0.1, s='Точка №4', fontweight='extra bold')
22 x, y = points[4]
23 ax13.text(x=x, y=y+0.1, s=r'$\mathscr{{P}}({0}, {1})$'.format(x, y))
24
25 ax13.text(x=0.2, y=1.2, s=r'$\lim_{x \to + \infty}\left(\frac{\pi(x)}{x/\ln
    ↪ x}\right)=1$', fontsize=20)
```

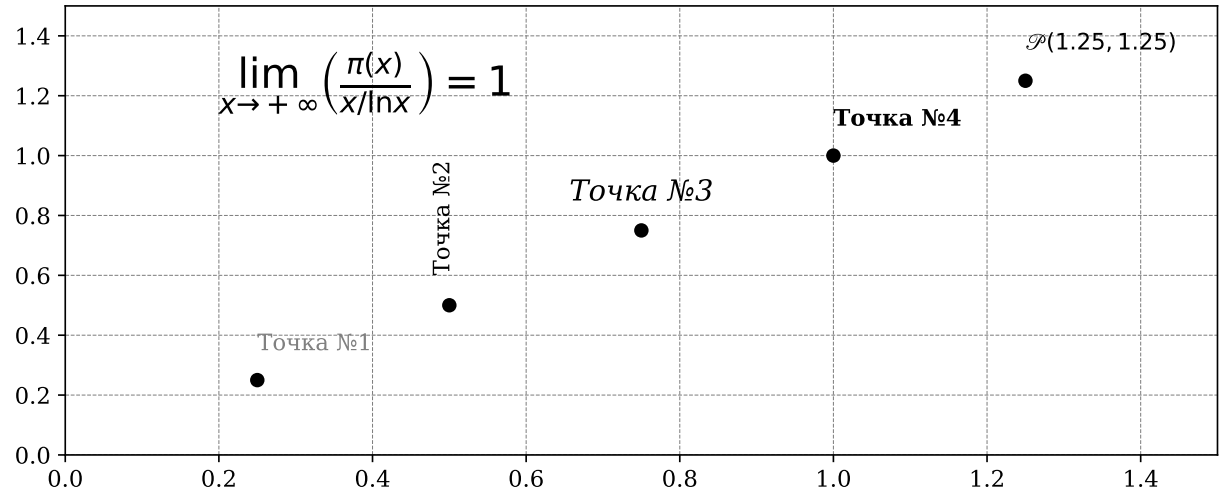

Рис. 3.15: Изображение `fig13`. Иллюстрирует текстовые средства `matplotlib` для добавления пояснений на изображение

В данном примере 3.15 показаны различные возможности по форматированию и преобразованию отображаемого на координатной плоскости текста. Для первой точки изменен цвет текста на серый. Для второй точки надпись повернута на 90 градусов вокруг точки (если не использовать `rotation_mode='anchor'`, то надпись будет повернута вокруг своего центра). У текста возле третей точки установлено выравнивание по центру, курсивное начертание и увеличенный размер шрифта. Для четвертой точки надпись выполнена жирным шрифтом. Наконец пятая точка обозначена рукописной буквой $\mathscr{P}$ с помощью команды LaTeX. Также отдельно на координатной плоскости записано математическое выражение в LaTeXнотации.

3.9.2. Аннотации

Аннотация представляет собой рамку с текстом и стрелку, которая отходит от рамки по направлению к аннотируемому объекту. Возможности по настройки внешнего вида аннотации практически безграничны, так как имеется возможность настраивать отдельно внешний вид текста, рамки и стрелки. Представление о всех возможностях можно получить на странице документации по ссылке `https://matplotlib.org/users/annotations.html`

```
fig14 = plt.figure(num=14)

ax14 = fig14.add_subplot(1,1,1)

ax14.set_xlim(0, 1.5)
ax14.set_ylim(0, 1.5)

points = [(0.25, 0.25), (0.50, 0.50), (0.75, 0.75), (1.0, 1.0), (1.25, 1.25)]

xs = [p for (p, _) in points]
ys = [p for (_, p) in points]

ax14.plot(xs, ys, marker='o', markersize=6, linestyle='None')

x, y = points[0]
ax14.annotate('Точка №1', xy=(x, y), xytext=(-25, 25), textcoords='offset
 ↪  points', arrowprops=dict(arrowstyle='->', lw=2, color='gray'),
 ↪  bbox=dict(boxstyle='round', fc='lightgray'))

x, y = points[1]
# Если не указать xytext, то аннотация будет расположенна прямо около точки
ax14.annotate('Точка №2', xy=(x, y), bbox=dict(boxstyle='round', fill=False))

x, y = points[2]
ax14.annotate('Точка №3', fontname='Comic Sans MS', xy=(x, y),
 ↪  horizontalalignment='center', xytext=(-100, 50), textcoords='offset
 ↪  points', arrowprops=dict(arrowstyle='->', lw=1, connectionstyle='angle3'),
 ↪  bbox=dict(boxstyle='circle', fc='lightgray'))

ax14.annotate('Точка №3', fontname='Consolas', xy=(x, y),
 ↪  horizontalalignment='center', xytext=(100, -95), textcoords='offset
 ↪  points', arrowprops=dict(arrowstyle='<->', lw=1, connectionstyle='angle'),
 ↪  bbox=dict(boxstyle='round4',fill=None))

x, y = points[3]
```

```
ax14.annotate('Точка №4', fontname='Mistral', xy=(x, y), fontsize=16,
    horizontalalignment='center', xytext=(100, -95), textcoords='offset
    points', arrowprops=dict(arrowstyle='-|>', linestyle='--'),
    bbox=dict(boxstyle='roundtooth', color='black', fill=None))

x, y = points[4]
ax14.annotate('Точка №4', xy=(x, y), xytext=(0, -70), textcoords='offset
    points', horizontalalignment='center', arrowprops=dict(arrowstyle='fancy',
    lw=0.5, fill=None), bbox=dict(boxstyle='darrow', lw=0.5, color='black',
    fill=None))
```

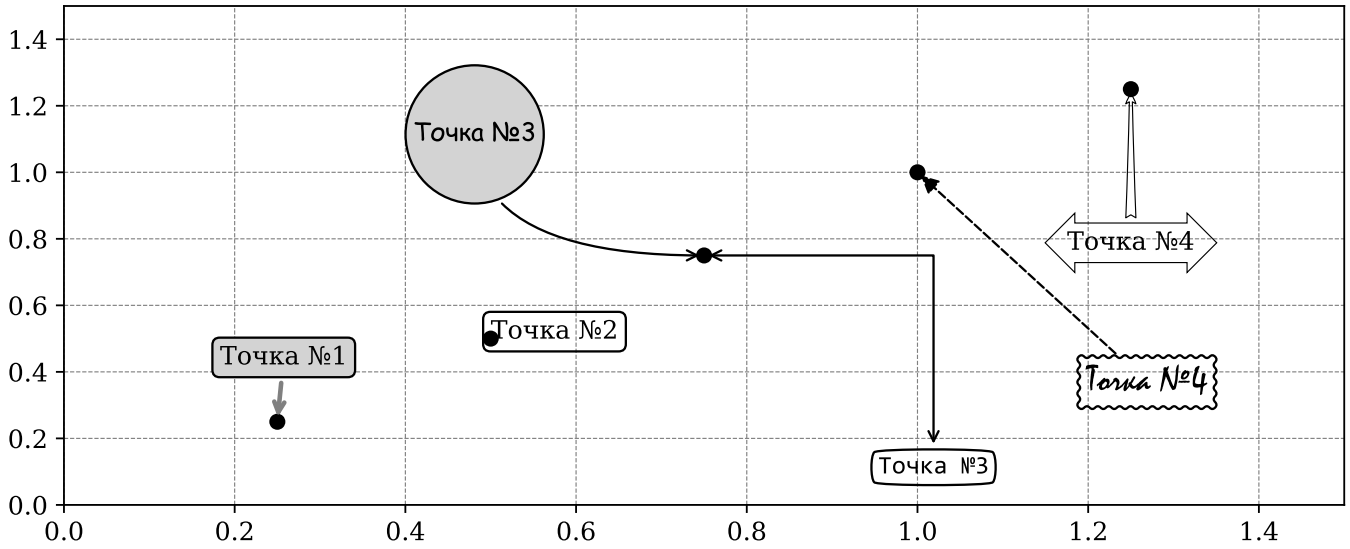

Рис. 3.16: Изображение `fig14`. Некоторые из большого числа возможных стилей аннотаций

3.10. Дополнительные средства для построения графиков

До сих пор во всех примерах для построения графиков мы пользовались лишь функцией `plot`. Данная функция благодаря гибкой настройке позволяет строить многообразные графики разного характера. Однако в библиотека `matplotlib` не ограничивается лишь одной этой функцией и предлагает большое разнообразие средств для визуализации, среди которых функции для построения гистограмм, спектров, круговых и столбчатых диаграмм, графиков с планками погрешностей, ступенчатых графиков, векторных полей, вертикальных и горизонтальных линий, диаграмм рассеяния и т.д. Описать их все здесь не представляется возможным, так что мы ограничимся лишь несколькими.

3.10.1. Построение поля векторов

Для построения поля векторов служит специальная функция `quiver` (буквально «колчан стрел», что иносказательно намекает на множество стрелок/векторов). Функцию `quiver` можно использовать для изображения как одного вектора, так и целого набора векторов, например касательных в каждой точке некоторой кривой. В качестве обязательных аргументов функция принимает два массива координат векторов по оси Ox (аргумент `U`) и по оси Oy (аргумент `V`). Следующие два аргумента — массивы `X` и `Y` — это наборы координат начала каждого из векторов. Если их опустить, то вектора будут распределены по равномерной сетке. В качестве примера изобразим касательные векторы для графика функции $\sin(x)$, параметрическое уравнение которой можно записать следующим образом:

$$\begin{cases} x = t, \\ y = \sin(t), \\ t \in \mathbb{R}, \end{cases} \quad \begin{cases} x' = 1, \\ y' = \cos(t), \\ t \in \mathbb{R}. \end{cases}$$

```
t = np.linspace(-2*np.pi, 2*np.pi, 50)
x = t
y = np.sin(t)
dx = np.ones(50)
dy = np.cos(t)

fig15 = plt.figure(num=15)
ax15 = fig15.add_subplot(1, 1, 1)

ax15.quiver(x[::4], y[::4], dx[::4], dy[::4], units='xy', angles='xy',
 ↪  width=0.05, scale=2.0, label='Касательные векторы')
ax15.plot(x, y, linewidth=0.5, color='gray', label=r'$\sin(t)$')

ax15.set_xlabel(r'Ось $Ox$')
ax15.set_ylabel(r'Ось $Oy$')
ax15.legend(loc=1)
ax15.set_aspect('equal')
fig15.savefig('img15.pdf', format='pdf', bbox_inches='tight', pad_inches=0)
```

3.10.2. Ступенчатый график

При отображении дискретных данных, часто есть необходимость в создании ступенчатых графиков. Например, если необходимо изобразить число заявок в некоторой системе обслуживания, то обычная функция `plot` с этой задачей не справится, так как будет пытаться соединить все точки прямыми отрезками, а не ступеньками. Этот факт иллюстрируется следующим примером.

```
fig16 = plt.figure(num=16, figsize=(10, 2))
ax16 = fig16.add_subplot(1, 1, 1)
```

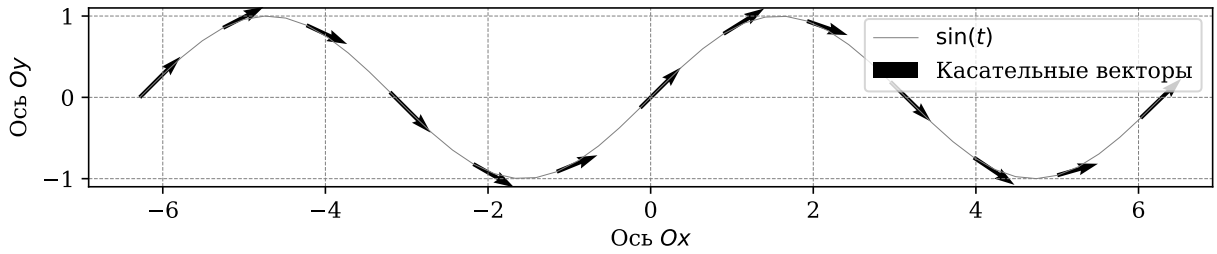

Рис. 3.17: Изображение `fig15`. Построение касательных векторов с помощью функции `quiver`

```

ax16.step([1, 2, 3, 4, 5, 6, 7, 8], [0, 0, 1, 1, 2, 3, 1, 1], linewidth=1.0,
 ↪  label='Функция step')
ax16.plot([1, 2, 3, 4, 5, 6, 7, 8], [0, 0, 1, 1, 2, 3, 1, 1], linewidth=1.0,
 ↪  linestyle='--', label='Функция plot')
ax16.legend()
fig16.savefig('img16.pdf', format='pdf', bbox_inches='tight', pad_inches=0)
```

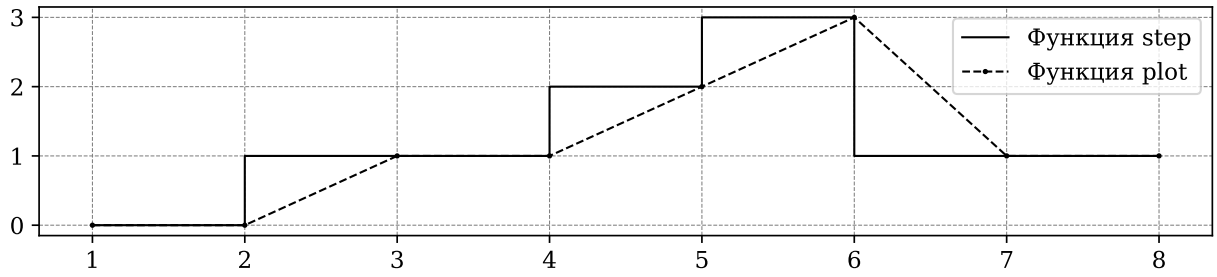

Рис. 3.18: Изображение `fig16`. Построение ступенчатого графика с помощью функции `step`

3.10.3. Горизонтальная и вертикальная прямые

Для изображения горизонтальной или вертикальной прямой можно обойтись и функцией `plot`, однако удобнее использовать специальные методы `hlines` и `vlines`. Пример ниже наглядно показывает как это сделать.

```
x = np.linspace(0, 1.25, 100)
y = np.sqrt(x)

fig17 = plt.figure(num=16, figsize=(10, 2))
ax17 = fig17.add_subplot(1, 1, 1)
ax17.set(xlim=(0.0, 1.25), ylim=(0.0, 1.5))

ax17.plot(x, y)

```

```
ax17.plot([0.25, 1.0, 1.0], [0.5, 0.5, 1.0], linestyle='None', marker='o',
 ↪  markersize=5)

ax17.hlines(y=1.0, xmin=0.0, xmax=1.0, linestyle='--', lw=1)
ax17.hlines(y=0.5, xmin=0.0, xmax=1.0, linestyle='--', lw=1)
ax17.vlines(x=0.25, ymin=0.0, ymax=0.5, linestyle='--', lw=1)
ax17.vlines(x=1.0, ymin=0.0, ymax=1.0, linestyle='--', lw=1)

ax17.text(0.6, 0.25, r'$\Delta x$', fontsize=16)
ax17.text(1.025, 0.65, r'$\Delta y = \sqrt{\Delta x}$', fontsize=16)

fig17.savefig('img17.pdf', format='pdf', bbox_inches='tight', pad_inches=0)
```

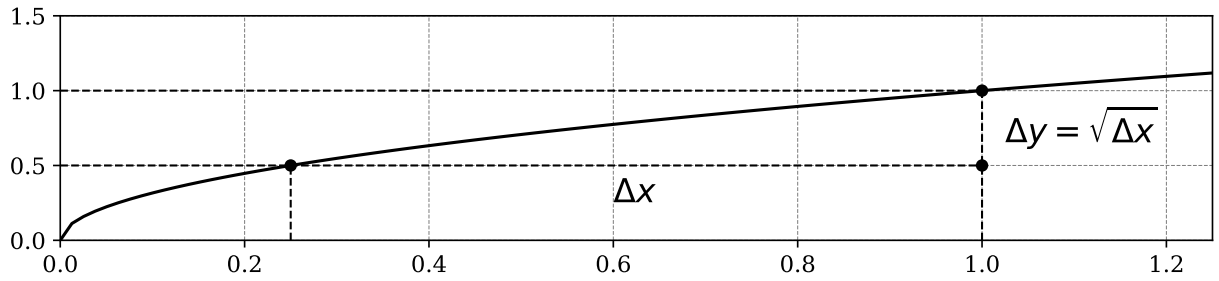

Рис. 3.19: Изображение `fig17`. Построение горизонтальной и вертикальной кривых.

3.11. Создание анимации с помощью ffmpeg

Рассмотрим способ создания анимированных кривых с помощью библиотеки `Matplotlib` и утилиты `ffmpeg`. Несмотря на то, что в `Matplotlib` встроены собственные средства для создания анимации, подход с использованием ffmpeg более универсален, так как его можно использовать в связке не только с `Matplotlib` но и с любыми другими библиотеками и утилитами для визуализации графиков.

Программа `ffmpeg` представляет собой утилиту, способную кодировать и декодировать видеофайлы, а также создавать видео из некоторого числа растровых изображений. Мы рассмотрим способ создания видеофайла в формате `.mp4` из набора `.png` файлов, которые будут генерироваться с помощью библиотеки `Matplotlib`.

Утилиту ffmpeg можно скачать на официальном сайте `https://www.ffmpeg.org/download.html`. Доступны сборки практически для всех операционных систем. Утилита представляет собой один исполняемый файл. В данном примере рассмотрим запуск `ffmpeg` под операционной системы Windows, однако он с минимальными изменениями может быть перенесен на случай Unix/GNU Linux и MacOS.

Кроме стандартных импортов библиотеки `NumPy` и `Matplotlib` необходимо импортировать подмодуль `mpatches`, который мы будем использовать для отрисовки окружности и два модуля стандартной библиотеки — `os` и `subprocess`. Модуль `os` позволяет получить доступ к стандартным операционной системы, в данном случае нам понадобится функция `mkdir`,

создающая пустую директорию. Модуль `subprocess` служит для фонового запуска внешних программ. Мы используем его для запуска `ffmpeg` непосредственно из `Python`-программы. Список импортов, таким образом, будет выглядеть следующим образом.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

import subprocess
import os
```

В качестве примера рассмотрим анимации циклоиды, которую описывает зафиксированная точка на катящейся окружности. Начнем с того, что создадим параметрическую функцию, задающую точки циклоиды:

$$\boldsymbol{r}(t) = \begin{cases} R(t - \sin t), \\ R(1 - \cos t), \end{cases}$$

где R — радиус генерирующей циклоиду окружности, а t — параметр, принимающий значения из $\mathbb{R}$.

```
def cycloid(t, R=3.0):
    '''Уравнение циклоиды'''
    return np.array([R*(t - np.sin(t)), R*(1 - np.cos(t))])

# укажем директорию, в которую будем
# сохранять сгенерированные картинки
FOLDER = 'cycloid'
try:
    os.mkdir(FOLDER)
except FileExistsError:
    pass

fig18 = plt.figure(1)
ax01 = fig18.add_subplot(1, 1, 1)
ax01.set_aspect('equal')
# радиус генерирующей окружности
R = 2.0
final_t = 10
final_x, _ = cycloid(final_t, R=R)

for counter, last_t in enumerate(np.arange(0.0, final_t + 0.1, 0.1)):
    # стираем все, что было на картинке
    ax01.clear()

```

```
    T = np.linspace(0.0, last_t, 1000)
    X, Y = cycloid(T, R=R)
    # последняя точка кривой
    last_x, last_y = cycloid(T[-1], R=R)

    # Генерирующая окружность с центром в точке (Rt, R)
    circ = mpatches.Circle((R*T[-1], R), R, fill=False, color='black')
    # Точка на катящейся окружности
    last_point = mpatches.Circle((last_x, last_y), 0.1, color='black')
    # задаем границы осей
    ax01.set_xlim(right=final_x + 2*R)
    ax01.set_ylim(top=2*R)
    # рисуем циклоиду
    ax01.plot(X, Y, linestyle='--')
    # добавляем на картинку точку и окружность
    ax01.add_patch(last_point)
    ax01.add_patch(circ)

    fig18.savefig('{0}/{1:03d}.png'.format(FOLDER, counter), dpi=300,
     ↪  format='png')
```

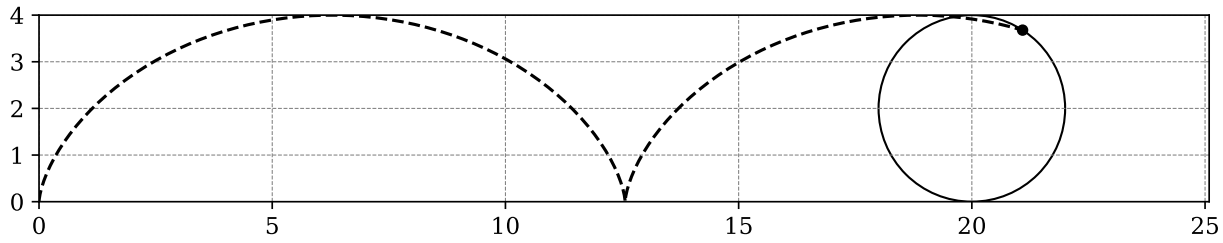

Рис. 3.20: Предпоследний кадр отрисовки циклоиды

После завершения работы программы, будет создана директория `cycloid`, куда будут сохранены все сгенерированные картинки. Для сборки картинок в видеофайл следует запустить `ffmpeg` в консоли со следующими опциями:

```
"G:\Program Files\ffmpeg\bin\ffmpeg.exe" -y -r 30 -f image2 -i cycloid\%03d.png
 ↪  -vcodec libx264 -crf 25 -pix_fmt yuv420p cycloid.mp4
```

Здесь указан полный путь до исполняемого файла `ffmpeg`, который зависит от способа установки и может различаться на разных компьютерах. Все остальные опции следует указать без изменения вне зависимости от операционной системы. Не будем вдаваться в подробности опций, укажем лишь, что `cycloid%03d.png` обозначает файлы в директории `cycloid` с именами из трех цифр: начиная от `000.png` и заканчивая `999.png`. В качестве последнего аргумента указано имя создаваемого видео-файла `cycloid`.

Следующий пример показывает, как запустить `ffmpeg` непосредственно из `Python` программы не открывая консоль.

```
FFMPEG = 'G:\Program Files\\ffmpeg\\bin\\ffmpeg.exe'
CMD = [FFMPEG, '-y', '-r', '30', '-f', 'image2', '-i',
 ↪ '{0}\%03d.png'.format(FOLDER), '-vcodec', 'libx264', '-crf', '25',
 ↪ '-pix_fmt', 'yuv420p', 'cycloid.mp4']
subprocess.run(CMD)
```

В результате будет создан видеофайл `cycloid.mp4` с анимацией катящейся окружности, точка которой отрисовывает линию циклоиды. Предпоследний кадр изображен на рисунке 3.20.

3.12. Вопросы для самопроверки

1. Для чего предназначена библиотека `Matplotlib`?
2. Какие преимущества есть у библиотеки `Matplotlib` по сравнению с конкурентами?
3. Какая магическая команда позволяет встраивать графики, построенные с помощью `Matplotlib` прямо в интерактивный блокнот `Jupyter`?
4. Как можно настроить внешний вид графиков, используемый `Matplotlib` по умолчанию?
5. Сколько осей координат может быть на одном изображении?
6. Что такое субграфик в терминах `Matplotlib`?
7. Какая функция является основной для построении плоских кривых?
8. Какие методы позволяют создавать сетки субкоординат (субграфиков)?
9. Какие параметры можно передавать функции `plot` для настройки внешнего вида строящихся кривых? Перечислите некоторые из них.
10. Как в `Jupyter` получить доступ к строке документации той или иной функции?
11. Какой метод следует вызвать, чтобы отобразить легенду для построенных графиков?
12. Какие настройки есть у легенды? Как добавить заголовок легенды? Как изменить ее расположение на рисунке?
13. Какие методы позволяют установить подписи к осям и заголовок?
14. Каким образом в текстовые элементы, располагаемые на осях можно добавить формулы в формате LaTeX?
15. Какие настройки регулируют цвет, размер и форму маркеров? Как настроить частоту расстановки маркеров на кривой?
16. Какие методы регулируют отображение символов разметки осей координат?

17. Какие методы позволяют манипулировать осями координат и регулировать их видимость?

18. Как построить график в полярной системе координат?

19. Что такое примитивы?

20. Какие примитивы есть в `Matplotlib`?

21. Какой метод позволяет добавить созданные примитивы на график?

22. Для чего нужен метод `Line2D`?

23. Какой метод позволяет добавлять на координатную плоскость текстовые метки?

24. Какие параметры, регулирующие внешний вид текста на координатной плоскости вы знаете?

25. Чем аннотации отличаются от обычного текста?

26. Может ли метод `annotate` заменить метод `text`?

27. Как настраиваются элементы аннотаций?

28. Какая функция позволяет визуализировать векторное поле?

29. Какая функция позволяет построить ступенчатый график?

30. Как в `Matplotlib` проще всего построить горизонтальную и вертикальную прямую линию?

31. Для чего нужна утилита `ffmpeg` и как её использовать в связке с `Matplotlib`?

32. Пользуясь дополнительными источниками и официальной документацией, ответьте на следующие вопросы.

    - Какая функция `Matplotlib` позволяет строить гистограммы?
    - Как установить логарифмический масштаб вдоль одной из осей координат?
    - Как минимизировать поля при сохранении созданного изображения в файл?
    - Какие функции `Matplotlib` позволяют строить трехмерные графики?
    - Есть ли в `Matplotlib` встроенные средства создания анимированных графиков?
    - Какие недостатки `Matplotlib` вы можете выделить после знакомства с этой библиотекой?

Глава 4

Компьютерная геометрия на плоскости

4.1. Предмет компьютерной геометрии

Компьютерная геометрия это раздел прикладной математики в котором изучается и разрабатывается математический аппарат, используемый в компьютерной (машинной) графике [20, 21, 22]. Основной задачей компьютерной геометрии является разработка способов эффективного вычисления точек плоских и пространственных кривых, а также поверхностей в трехмерном пространстве. Также изучаются различные геометрические преобразования на плоскости и в пространстве, такие как вращение, перенос, изменение масштаба и т.д.

Результаты компьютерной геометрии используются в самых разных областях человеческой деятельности, связанной с применением современных компьютеров. В качестве примеров можно привести системы автоматизированного проектирования (CAD — computer aided design, CAM — computer aided manufacturing), в векторные графические редакторы (Adobe Illustrator, CorelDRAW, Inkscape), векторные форматы изображений (SVG, PDF, PS), издательское дело (векторные шрифты, например стандарты TrueType и OpenType), веб-программирование (технология Canvas) компьютерные игры (DirectX и OpenGL) и компьютерную трехмерную анимацию, которая в настоящее время сильно потеснила классическую рисованную анимацию.

4.2. Пространственные и плоские кривые

В связи с ограниченностью данного курса одним семестром, основным предметом изучения будут служить плоские и пространственные кривые и способы их построения с помощью компьютера. В данном разделе мы кратко изложим сведения из аналитической и дифференциальной геометрии, необходимые для понимания дальнейшего материала. За более подробным изложением следует обратиться к учебным пособиям по этому предмету [23, 24, 25, 26, 27, 28, 29, 30].

4.2.1. Параметрическое уравнение кривой

Геометрическое место точек, радиус-векторы $\boldsymbol{r}$ которых определяются уравнением

$$\boldsymbol{r} = \boldsymbol{r}(t),\ t \in [a,b] \subset \mathbb{R},$$

представляет *кривую* (или, более точно, регулярный кусок кривой), если в этом интервале функция $\boldsymbol{r}(t)$ непрерывна и имеет непрерывную производную $\dot{\boldsymbol{r}}(t)$, не обращающуюся в ноль внутри отрезка $[a,b]$:

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} \neq 0,\ \forall t \in [a,b].$$

В евклидовом пространстве E^n с базисом $\langle \boldsymbol{e}_1, \dots, \boldsymbol{e}_n \rangle$ кривая представима в виде системы из n непрерывных и дифференцируемых функций аргумента t:

$$\boldsymbol{r} = \sum_{i=1}^{n} x^i(t)\boldsymbol{e}_i = x^1(t)\boldsymbol{e}_1 + \dots + x^n(t)\boldsymbol{e}_n$$

В частности, в трехмерном евклидовом пространстве в декартовых координатах $\boldsymbol{r}(t) = x(t)\boldsymbol{e}_1 + y(t)\boldsymbol{e}_2 + z(t)\boldsymbol{e}_3$, где $\boldsymbol{e}_1 = (1,0,0)^T$, $\boldsymbol{e}_2 = (0,1,0)^T$ и $\boldsymbol{e}_3 = (0,0,1)^T$ и

$$\boldsymbol{r}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

Производная от векторозначной функции $\boldsymbol{r}(t)$ в произвольной точке

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \left(\frac{\mathrm{d}x^1}{\mathrm{d}t}, \dots, \frac{\mathrm{d}x^n}{\mathrm{d}t} \right)^T = (\dot{x}^1, \dots, \dot{x}^n)^T,$$

называется *касательным вектором* и устанавливает положительное направление касательной в точке P к кривой, задаваемой функцией $\boldsymbol{r}$.

Одну и туже кривую можно параметризовать разными способами. Рассмотрим параметр $l = l(t)$, называемый *натуральным*, при котором касательный вектор, получающийся при дифференцировании функции по этому параметру, будет единичным вектором при любых значениях l:

$$\left| \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l} \right| \equiv 1, \forall l \in [a,b].$$

Чтобы найти связь l и t, продифференцируем $\boldsymbol{r}(l)$ как сложную функцию $\boldsymbol{r}(l(t))$

$$\frac{\mathrm{d}\boldsymbol{r}(l(t))}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\frac{\mathrm{d}l}{\mathrm{d}t} \Rightarrow \left|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}\right| = \left|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\frac{\mathrm{d}l}{\mathrm{d}t}\right| = \frac{\mathrm{d}l}{\mathrm{d}t}\left|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\right| = \frac{\mathrm{d}l}{\mathrm{d}t}.$$

В результате найдем связь между $\mathrm{d}t$ и $\mathrm{d}l$:

$$\mathrm{d}l = \|\dot{\boldsymbol{r}}(t)\|\mathrm{d}t = \left\|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}\right\|\mathrm{d}t = \sqrt{\left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}, \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}\right)}\mathrm{d}t.$$

В частности в случае пространства E^3 можно записать

$$\mathrm{d}l = \sqrt{\dot{x}^2(t) + \dot{y}^2(t) + \dot{z}^2(t)}\mathrm{d}t.$$

Геометрический смысл натурального параметра l заключается в следующем. Мы фиксируем на кривой некоторую точку O и принимаем ее за начало отсчета. Любую другую точку P кривой можно однозначно определить, как расстояние l пройденное по кривой от точки O до точки P. Положительные и отрицательные значение l соответствуют разным направлениям перемещения по кривой. Заметим, что если удастся проинтегрировать выражение $\|\dot{\boldsymbol{r}}\|\mathrm{d}t$, то будет найдена связь между t и l с точностью до константы. Константа как раз отражает произвол в выборе точки начала отсчета.

Кривая, которая допускает введение натурального параметра и, как следствие, длины дуги, называется *спрямляемой*. Кривая спрямляема, если текущие координаты являются непрерывными функциями параметра t с непрерывными первыми производными. Натуральный параметр l на кривой, для которого производная от радиус-вектора становится единичным вектором, есть длина дуги кривой, измеряемая от произвольно, но определенно выбранного начала отсчета дуги на кривой:

$$l = \int_c^t \sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2}\mathrm{d}t.$$

4.2.2. Репер Френе трехмерной и плоской кривых

Вторая производная от векторозначной функции $\boldsymbol{r}(l)$ в точке $P = \boldsymbol{r}(l_0) = (x^1(l_0), \dots, x^n(l_0))$

$$\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}(l_0) = \ddot{\boldsymbol{r}}(t) = (\ddot{x}^1(l_0), \dots, \ddot{x}^n(l_0)),$$

называется *вектором нормали* и устанавливает положительное направление нормали в точке P к кривой, задаваемой функцией $\boldsymbol{r}(t)$.

Векторы касательной и нормали ортогональны, если параметр l натуральный, что видно из следующих формул:

$$\frac{\mathrm{d}}{\mathrm{d}l}\left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}, \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\right) = \left(\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}, \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\right) + \left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}, \frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}\right) = 2\left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}, \frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}\right),$$

$$\frac{\mathrm{d}}{\mathrm{d}l}\left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}, \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\right) = \frac{\mathrm{d}}{\mathrm{d}l}\left|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}\right|^2 = \frac{\mathrm{d}}{\mathrm{d}l}1 = 0,$$

$$\left(\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}l}, \frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}\right) = 0.$$

Кривизной кривой $\boldsymbol{r}(l)$ называется длина вектора нормали k

$$k = \left|\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}\right|,$$

а *радиусом кривизны* называется число, обратное значению кривизны $R = 1/k$.

Обычно используют *единичный вектор нормали* $\boldsymbol{n}$, который определяется с помощью формулы

$$\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2} = k\boldsymbol{n},\ \boldsymbol{n} = \frac{\dfrac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}}{\left|\dfrac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}l^2}\right|}$$

Касательный вектор имеет физический смысл вектора скорости, а нормальный вектор — вектора ускорения.

Вектор $\boldsymbol{b} = [\boldsymbol{v}, \boldsymbol{n}]$ называется вектором *бинормали*. по определению векторного умножения вектор бинормали ортогонален векторам $\boldsymbol{v}$ и $\boldsymbol{n}$. Тройка векторов $\langle \boldsymbol{v}, \boldsymbol{n}, \boldsymbol{b} \rangle$ образует репер, который называется *репером Френе*.

$$[\boldsymbol{v}, \boldsymbol{n}] = \boldsymbol{b},\ [\boldsymbol{n}, \boldsymbol{b}] = \boldsymbol{v},\ [\boldsymbol{b}, \boldsymbol{v}] = \boldsymbol{n}.$$

Для любой пространственной кривой $\boldsymbol{r}(l)$, где l — натуральный параметр, имеют место следующие формулы, называемые *формулами Френе*:

$$\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}l} = k\boldsymbol{n},\ \frac{\mathrm{d}\boldsymbol{n}}{\mathrm{d}l} = -k\boldsymbol{v} - \kappa\boldsymbol{b},\ \frac{\mathrm{d}\boldsymbol{b}}{\mathrm{d}l} = \kappa\boldsymbol{n}$$

где $\boldsymbol{v}$ — вектор касательной, $\boldsymbol{n}$ — вектор нормали, $\boldsymbol{b}$ — вектор бинормали, κ — *кручение*, а k — кривизна.

В случае плоской кривой репер Френе состоит только из двух векторов: касательного и нормального. Приведем формулы для компонент этих векторов в декартовых координатах для плоской кривой общего вида.

Плоская кривая в декартовыой системе координат в параметрическом виде задается следующей формулой:

$$\boldsymbol{r}(t) = (x(t), y(t))^T,$$

а касательный и нормальный векторы вычисляются по следующим формулам:

$$\boldsymbol{v} = \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = (\dot{x}(t), \dot{y}(t)),\ \boldsymbol{n} = \frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}t^2} = (\ddot{x}(t), \ddot{y}(t)).$$

Репер Френе образуют нормированные векторы:

$$\boldsymbol{v}_0 = \frac{\boldsymbol{v}}{|\boldsymbol{v}|} = \left(\frac{\dot{x}}{\sqrt{\dot{x}^2 + \dot{y}^2}}, \frac{\dot{y}}{\sqrt{\dot{x}^2 + \dot{y}^2}}\right),\ \boldsymbol{n}_0 = \frac{\mathrm{d}\boldsymbol{v}_0/\mathrm{d}t}{|\mathrm{d}\boldsymbol{v}_0/\mathrm{d}t|}.$$

Следует обратить внимание на то, что для вычисления нормированного нормального вектора надо взять первую производную от нормированного вектора $\boldsymbol{v}_0$, а не вторую производную от радиус-вектора $\boldsymbol{r}(t)$.

Явное выражение компонент нормального вектора через функции $x(t)$ и $y(t)$ имеет простой вид.

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\dot{x}}{\sqrt{\dot{x}^2+\dot{y}^2}} = \frac{\dot{y}(\dot{y}\ddot{x}-\dot{x}\ddot{y})}{(\sqrt{\dot{x}^2+\dot{y}^2})^3},\quad \frac{\mathrm{d}}{\mathrm{d}t}\frac{\dot{y}}{\sqrt{\dot{x}^2+\dot{y}^2}} = \frac{\dot{x}(\dot{x}\ddot{y}-\dot{y}\ddot{x})}{(\sqrt{\dot{x}^2+\dot{y}^2})^3},\quad \left|\frac{\mathrm{d}\boldsymbol{v}_0}{\mathrm{d}t}\right| = \frac{\dot{x}\ddot{y}-\dot{y}\ddot{x}}{\dot{x}^2+\dot{y}^2}$$

$$\boldsymbol{n}_0 = \frac{\mathrm{d}\boldsymbol{v}_0}{\mathrm{d}t} : \left|\frac{\mathrm{d}\boldsymbol{v}_0}{\mathrm{d}t}\right| = \left(\frac{-\dot{y}}{\sqrt{\dot{x}^2+\dot{y}^2}}, \frac{\dot{x}}{\sqrt{\dot{x}^2+\dot{y}^2}}\right)$$

Эволютой кривой называют геометрическое место точек, являющихся центрами кривизны данной кривой. *Эвольвентой* (инволютой) кривой в свою очередь называется кривая, нормаль в каждой точки которой является касательной к исходной кривой. По отношению к своей эволюте любая кривая является эвольвентой.

Зная явные формулы для касательного и нормального векторов в декартовой системе координат, можно получить явные формулы для радиус вектора эволюты. Центр кривизны $\boldsymbol{M}$ лежит на главной нормали на расстоянии радиуса кривизны от кривой:

$$\boldsymbol{M} = \boldsymbol{r} + R\boldsymbol{n}.$$

Радиус кривизны в декартовой системе координат вычисляется по формуле:

$$R = \frac{(\dot{x}^2+\dot{y}^2)^{3/2}}{\dot{x}\ddot{y}-\dot{y}\ddot{x}},$$

поэтому радиус-вектор эволюты имеет вид:

$$\boldsymbol{M} = \begin{pmatrix} x - \dot{y}\dfrac{\dot{x}^2+\dot{y}^2}{\dot{x}\ddot{y}-\dot{y}\ddot{x}} \\ y - \dot{x}\dfrac{\dot{x}^2+\dot{y}^2}{\dot{x}\ddot{y}-\dot{y}\ddot{x}} \end{pmatrix}$$

4.3. Сплайны

Во многих практических задачах аналитическая форма некоторой двумерной или трехмерной кривой не известна и ее необходимо восстановить по конечному набору точек и ряду дополнительных условий, налагаемых на искомую кривую в этих точках. Данная задача называется задачей *интерполяции кривой* по заданным точкам. Наиболее очевидный подход к этой задаче — соединить известные точки отрезками прямых и получить некоторую *ломанную линию*. Однако чаще всего полученная ломанная плохо отражает свойства искомой кривой и используется лишь для грубого приближения.

Более точные результаты дает интерполяция с помощью *сплайнов*. Сплайном изначально называлась гибкая линейка с помощью которой проводили соединительные кривые при

построении чертежей на бумаге. Математический же сплайн — это кусочный полином степени n. На сплайн часто налагают дополнительные условия, вроде условия существования первых и вторых производный в точках соединения сегментов. Степень n обычно мала (от 2 до 5), так как для полиномов высоких степеней вероятно возникновение *феномена Рунге*.

Задача ставится следующим образом: имеется совокупность точек на плоскости или в пространстве, радиус-векторы которых равны $\boldsymbol{p}_i = (\boldsymbol{p}_1, \dots, \boldsymbol{p}_n)$, $i = 0, \dots, n$. Требуется построить линию, радиус-вектор которой при значениях параметра t_i был бы равен $\boldsymbol{p}_i$.

$$\boldsymbol{r}(t_0) = \boldsymbol{p}_0, \dots, \boldsymbol{r}(t_i) = \boldsymbol{p}_i, \dots, \boldsymbol{r}(t_n) = \boldsymbol{p}_n.$$

Точки $\boldsymbol{p}_i$ называются *опорными* или *характеристическими* точками кривой, а значения параметра t_i — *узловыми* точками.

Кроме самих точек, могут быть заданы производные первого $\dot{\boldsymbol{p}}_i = \dot{\boldsymbol{r}}(t_i)$ и второго порядка $\ddot{\boldsymbol{p}}_i = \ddot{\boldsymbol{r}}(t_i)$.

4.3.1. Ломанная

Простейшая точечно-заданная линия называется *ломанной линией*. Она состоит из отрезков, последовательно соединяющих заданные точки. Значения параметра в каждой последующей точке больше предыдущего: $t_i < t_{i+1}$. Радиус вектор ломанной определяется равенством:

$$\boldsymbol{r}(t) = \boldsymbol{p}_i(1 - \tau) + \boldsymbol{p}_{i+1}\tau, \quad \tau = \frac{t - t_i}{t_{i+1} - t_i}, \quad t_0 \le t \le t_n.$$

Параметр τ называется внутренним и он, по определению, нормирован так, что изменяется в пределах отрезка $[0, 1]$. Взяв две опорные точки $\boldsymbol{p}_i$ и $\boldsymbol{p}_{i+1}$ и последовательно изменяя параметр от $[0, 1]$ мы получим все точки отрезка прямой линии, соединяющего точки $\boldsymbol{p}_i$ и $\boldsymbol{p}_{i+1}$.

Большинство средств для построения графиков по заданным точкам фактически проводят интерполяцию с помощью ломанной линии. Если известно большое количество близко расположенных точек, то интерполяция ломанной может дать очень хорошее приближение к реальной кривой.

4.3.2. Интерполяция параметрическими полиномами высокого порядка

При использовании для интерполяции полиномов, очевидным решением является выбор полинома максимально возможного порядка для достижения максимально возможной точности. Пусть даны узловые точки $t_0, t_1, t_2, \dots, t_{n-1}, t_n$ и соответствующие опорные точки $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$,..., $\boldsymbol{p}_n$. Имеется возможность построить параметрический полином $\boldsymbol{r}(t)$ порядка не выше n:

$$\boldsymbol{r}(t) = \boldsymbol{a}_n t^n + \boldsymbol{a}_{n-1} t^{n-1} + \dots + \boldsymbol{a}_1 t + \boldsymbol{a}_0.$$

Для нахождения коэффициентов полинома необходимо решить следующую систему уравнений:

$$\begin{cases} \boldsymbol{r}(t_0) = \boldsymbol{p}_0, \\ \boldsymbol{r}(t_1) = \boldsymbol{p}_1, \\ \vdots \\ \boldsymbol{r}(t_n) = \boldsymbol{p}_n, \end{cases}$$

которая в матричном виде записывается следующим образом:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & \dots & t_0^{n-1} & t_0^n \\ 1 & t_1 & t_1^2 & \dots & t_1^{n-1} & t_1^n \\ 1 & t_2 & t_2^2 & \dots & t_2^{n-1} & t_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & t_{n-1} & t_{n-1}^2 & \dots & t_{n-1}^{n-1} & t_{n-1}^n \\ 1 & t_n & t_n^2 & \dots & t_n^{n-1} & t_n^n \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_{n-1} \\ \boldsymbol{a}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{p}_{n-1} \\ \boldsymbol{p}_n \end{bmatrix}$$

где переменные и правая часть представляют собой матрицы $3 \times (n+1)$ (для плоской кривой $2 \times (n+1)$):

$$\begin{bmatrix} \boldsymbol{a}_0 \\ \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_{n-1} \\ \boldsymbol{a}_n \end{bmatrix} = \begin{bmatrix} a_0^x & a_0^y & a_0^z \\ a_1^x & a_1^y & a_1^z \\ a_2^x & a_2^y & a_2^z \\ \vdots & \vdots & \vdots \\ a_{n-1}^x & a_{n-1}^y & a_{n-1}^z \\ a_n^x & a_n^y & a_n^z \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{p}_{n-1} \\ \boldsymbol{p}_n \end{bmatrix} = \begin{bmatrix} p_0^x & p_0^y & p_0^z \\ p_1^x & p_1^y & p_1^z \\ p_2^x & p_2^y & p_2^z \\ \vdots & \vdots & \vdots \\ p_{n-1}^x & p_{n-1}^y & p_{n-1}^z \\ p_n^x & p_n^y & p_n^z \end{bmatrix}$$

После нахождения коэффициентов $\{\boldsymbol{a}_i\}_0^n$, можно использовать параметрический полином для интерполяции искомой кривой. Однако, вопреки ожиданиям, увеличение порядка полинома необязательно приводит к повышению точности интерполяции. Для определенных кривых увеличение порядка интерполяционного полинома приводит к возрастанию погрешностей интерполяции в окрестности концов промежутка интерполяции. Этот феномен получил название *феномена Рунге*. Его проявление можно проиллюстрировать на примере *функции Рунге* $y(x) = 1/(1+x^2)$. График интерполяционного полинома для функции Рунге изображен на рисунке 4.1. Из графика можно видеть, что на концах промежутка интерполяционный полином претерпевает колебания, амплитуда которых будет возрастать при росте порядка полинома. При это в центральной части области точность интерполяции растет.

Чтобы избежать возникновения феномена Рунге, используют кусочную интерполяцию полиномами низких порядков.

4.3.3. Сплайн Лагранжа

Сплайн Лагранжа для опорных точек $\{\boldsymbol{p}_i\}_0^N$ строится по следующей формуле:

$$\boldsymbol{r}(t) = \sum_{i=0}^{N} L_i(t)\boldsymbol{p}_i, \quad L_i(t) = \prod_{j=0, j\neq i}^{N} \frac{t - t_j}{t_i - t_j}, \quad t \in [t_0, t_N].$$

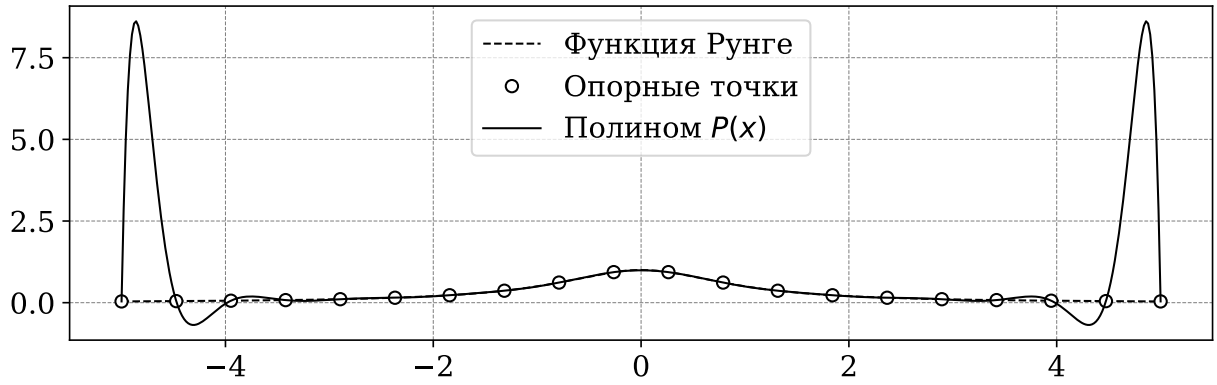

Рис. 4.1: Иллюстрация феномена Рунге на примере полинома 19-го порядка

Никаких дополнительных требований не накладывается и для построения необходимо задать лишь набор опорных точек. Функции $L_i(t)$ являются базисными функциями для полинома Лагранжа. Вышеизложенная формула подразумевает, что построение производится по всему массиву опорных точек сразу. Из-за чего, с увеличением числа точек, растет порядок полинома, что может привести к возникновению эффекта Рунге на концах интерполируемого отрезка.

Сплайн Лагранжа, таким образом, является одним из самых простых сплайнов и подходит для применения лишь в несложных и учебных задачах. Далее мы рассмотрим сплайны Эрмита и кубический сплайн.

4.3.4. Кубический сплайн

Кубическим сплайном принято называть параметрический сплайн, который в точках соединения своих сегментов имеет непрерывные первые и вторые производные. Уравнение одного сегмента такого сплайна имеет вид:

$$\boldsymbol{r}_i(t) = \boldsymbol{a}_{i0} + \boldsymbol{a}_{i1}t + \boldsymbol{a}_{i2}t^2 + \boldsymbol{a}_{i3}t^3, \;\; t_i \leqslant t \leqslant t_{i+1}, \; i = 0, \dots, N.$$

где для трехмерного случая коэффициенты $\boldsymbol{a}_{i0}$, $\boldsymbol{a}_{i1}$, $\boldsymbol{a}_{i2}$ и $\boldsymbol{a}_{i3}$ имеют следующий вид:

$$\boldsymbol{a}_{i0} = \begin{bmatrix} a_{0x} \\ a_{0y} \\ a_{0z} \end{bmatrix} \boldsymbol{a}_{i1} = \begin{bmatrix} a_{1x} \\ a_{1y} \\ a_{1z} \end{bmatrix} \boldsymbol{a}_{i2} = \begin{bmatrix} a_{2x} \\ a_{2y} \\ a_{2z} \end{bmatrix} \boldsymbol{a}_{i3} = \begin{bmatrix} a_{3x} \\ a_{3y} \\ a_{3z} \end{bmatrix}$$

Без потери общности можно положить $0 \leqslant t \leqslant t_i$ для каждого $i = 1, \dots, N$. В этом случае говорят о *хордовой* интерполяции с помощью кубических сплайнов.

Для построения каждого сегмента сплайна необходимы две опорные точки $\boldsymbol{p}_i$, $\boldsymbol{p}_{i+1}$ и два касательных вектора $\dot{\boldsymbol{p}}_i$ и $\dot{\boldsymbol{p}}_{i+1}$. На основе опорных точек и касательных векторов можно определить коэффициенты $\boldsymbol{a}_{i0}$, $\boldsymbol{a}_{i1}$, $\boldsymbol{a}_{i2}$, $\boldsymbol{a}_{i3}$ (обратите внимание, что у каждого сегмента сплайна эти коэффициенты могут быть разными).

Рассмотрим вначале только первый сегмент сплайна $\boldsymbol{r}_0(t)$. Для его построения должны быть заданы две точки $\boldsymbol{p}_0, \boldsymbol{p}_1$ (радиус-векторы точек) и касательные векторы $\dot{\boldsymbol{p}}_0, \dot{\boldsymbol{p}}_1$. Опустим

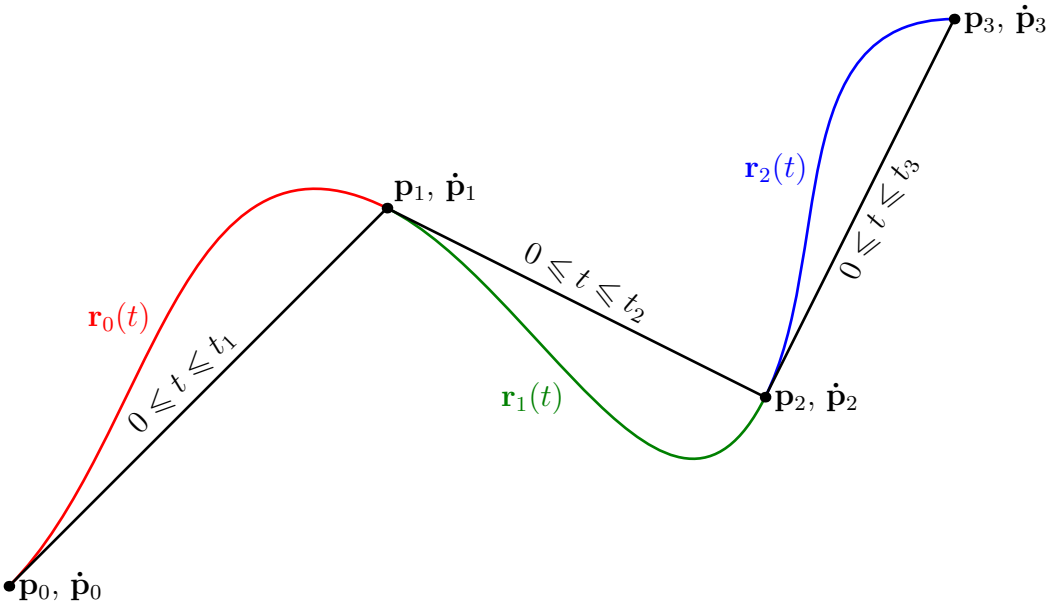

Рис. 4.2: Три сегмента $\boldsymbol{r}_0(t)$, $\boldsymbol{r}_1(t)$, $\boldsymbol{r}_2(t)$ кубического сплайна при хордовой интерполяции, где $t_1 = |\boldsymbol{p}_1 - \boldsymbol{p}_0|$, $t_2 = |\boldsymbol{p}_2 - \boldsymbol{p}_1|$, $t_3 = |\boldsymbol{p}_3 - \boldsymbol{p}_2|$

индексы i у коэффициентов и составим систему уравнений, из которой эти коэффициенты можно будет найти.

$$\begin{cases} \boldsymbol{r}(0) = \boldsymbol{p}_0, \\ \boldsymbol{r}(t_1) = \boldsymbol{p}_1, \\ \dot{\boldsymbol{r}}(0) = \dot{\boldsymbol{p}}_0, \\ \dot{\boldsymbol{r}}(t_1) = \dot{\boldsymbol{p}}_1. \end{cases} \Rightarrow \begin{cases} \boldsymbol{a}_0 = \boldsymbol{p}_0, \\ \boldsymbol{a}_0 + \boldsymbol{a}_1 t_1 + \boldsymbol{a}_2 t_1^2 + \boldsymbol{a}_3 t_1^3 = \boldsymbol{p}_1, \\ \boldsymbol{a}_1 = \dot{\boldsymbol{p}}_0, \\ \boldsymbol{a}_1 + 2\boldsymbol{a}_2 t_1 + 3\boldsymbol{a}_3 t_1^2 = \dot{\boldsymbol{p}}_1. \end{cases}$$

Решая систему относительно $\boldsymbol{a}_0$, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$ получим следующие выражения:

$$\begin{cases} \boldsymbol{a}_0 = \boldsymbol{p}_0, \\ \boldsymbol{a}_0 = \dot{\boldsymbol{p}}_0, \\ \boldsymbol{a}_2 = \dfrac{3(\boldsymbol{p}_1 - \boldsymbol{p}_0)}{t_1^2} - \dfrac{2\dot{\boldsymbol{p}}_0}{t_1} - \dfrac{\dot{\boldsymbol{p}}_1}{t_1}, \\ \boldsymbol{a}_3 = \dfrac{2(\boldsymbol{p}_0 - \boldsymbol{p}_1)}{t_1^3} + \dfrac{\dot{\boldsymbol{p}}_0}{t_1^2} + \dfrac{\dot{\boldsymbol{p}}_1}{t_1^2}. \end{cases} \tag{4.1}$$

Аналогичные выражения получаются и для все остальных сегментов сплайна, соединяющих точки $\boldsymbol{p}_2$, $\boldsymbol{p}_3$, $\boldsymbol{p}_4$ и т.д. При хордовой интерполяции параметр t для каждого сегмента принимает значения из отрезка $[0, t_{i+1}]$. Значение t_{i+1} имеет геометрический смысл длины хорды (отрезка прямой), соединяющей соседние опорные точки (см. рис. 4.2):

$$t_{i+1} = \|\boldsymbol{p}_{i+1} - \boldsymbol{p}_i\|.$$

Так, для трехмерного случая:

$$\begin{aligned} t_1 &= \|\boldsymbol{p}_1 - \boldsymbol{p}_0\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}, \\ t_2 &= \|\boldsymbol{p}_2 - \boldsymbol{p}_1\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}, \\ &\dots \\ t_N &= \|\boldsymbol{p}_N - \boldsymbol{p}_{N-1}\| = \sqrt{(x_N - x_{N-1})^2 + (y_N - y_{N-1})^2 + (z_N - z_{N-1})^2}. \end{aligned} \tag{4.2}$$

При известных производных $\dot{\boldsymbol{p}}_i$ для построения всех сегментов было бы достаточно пользоваться формулами (4.1). Однако чаще всего значения производных $\dot{\boldsymbol{p}}_i$ не известны и их приходится вычислять дополнительно, что можно сделать используя условие непрерывности вторых производных в каждой точке кубического сплайна.

Так, например, вторая производная первого сегмента $\ddot{\boldsymbol{r}}_0(t)$ в точке $\boldsymbol{p}_1$ должна совпадать с производной второго сегмента $\ddot{\boldsymbol{r}}_1(t)$ в этой же точке, то есть должно выполняться следующее равенство:

$$\frac{\mathrm{d}^2\boldsymbol{r}_0}{\mathrm{d}t^2}\Big|_{t=t_1} = \frac{\mathrm{d}^2\boldsymbol{r}_1}{\mathrm{d}t^2}\Big|_{t=0} \quad \text{или} \quad \ddot{\boldsymbol{r}}_0(t_1) = \ddot{\boldsymbol{r}}_1(0).$$

Записав такие равенства для всех точек соединения сегментов, можно получить систему линейных уравнений, которая позволит вычислить касательные вектора во всех этих точках, кроме начальной $\boldsymbol{p}_0$ и конечной $\boldsymbol{p}_N$. Подробные выкладки можно найти в книге [21], здесь приведем лишь основные формулы.

Значения касательных векторов $\dot{\boldsymbol{p}}_i$ находятся из следующей системы уравнений:

$$\boldsymbol{M} \cdot \dot{\boldsymbol{P}} = \boldsymbol{R}, \tag{4.3}$$

где матрица $\boldsymbol{M}$ имеет размерность $(N+1) \times (N+1)$ и является трехдиагональной:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ t_2 & 2(t_1+t_2) & t_1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & t_3 & 2(t_2+t_3) & t_2 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & t_4 & 2(t_3+t_4) & t_3 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & t_{N-1} & 2(t_{N-1}+t_{N-2}) & t_{N-2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & t_N & 2(t_N+t_{N-1}) & t_{N-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 \end{bmatrix}$$

Обратите внимание, что размер матрицы зависит от количества точек $\boldsymbol{p}_i$ или, иначе, от количества сегментов сплайна. При применении хордовой интерполяции элементы матрицы вычисляются по формулам (4.2) на основе заданных точек $\boldsymbol{p}_i$.

Матрица $\dot{\boldsymbol{P}}$ имеет размерность $(N+1) \times 3$ для пространственных кривых ($(N+1) \times 2$ для плоских кривых) и состоит из компонент касательных векторов:

$$\dot{\boldsymbol{P}} = \begin{bmatrix} \dot{x}_0 & \dot{y}_0 & \dot{z}_0 \\ \dot{x}_1 & \dot{y}_1 & \dot{z}_1 \\ \vdots & \vdots & \vdots \\ \dot{x}_N & \dot{y}_N & \dot{z}_N \end{bmatrix} \text{ или } \begin{bmatrix} \dot{x}_0 & \dot{y}_0 \\ \dot{x}_1 & \dot{y}_1 \\ \vdots & \vdots \\ \dot{x}_N & \dot{y}_N \end{bmatrix}$$

Относительно этой матрицы и необходимо решать данную линейную систему линейных уравнений. В библиотеки NumPy для решения таких систем предусмотрена функция `np.linalg.solve(M,` Наконец матрица $\boldsymbol{R}$ имеет следующий вид:

$$\boldsymbol{R} = \begin{bmatrix} \dot{\boldsymbol{p}}_0 \\ \frac{3}{t_1 t_2}(t_1^2(\boldsymbol{p}_2 - \boldsymbol{p}_1) + t_2^2(\boldsymbol{p}_1 - \boldsymbol{p}_0)) \\ \frac{3}{t_2 t_3}(t_2^2(\boldsymbol{p}_3 - \boldsymbol{p}_2) + t_3^2(\boldsymbol{p}_2 - \boldsymbol{p}_1)) \\ \vdots \\ \frac{3}{t_{N-1} t_N}(t_{N-1}^2(\boldsymbol{p}_N - \boldsymbol{p}_{N-1}) + t_N^2(\boldsymbol{p}_{N-1} - \boldsymbol{p}_{N-2}))\boldsymbol{p}_N \\ \dot{\boldsymbol{p}}_N \end{bmatrix}$$

и также состоит из трех столбцов для трехмерного из двух для двумерного случая.

Касательные векторы $\dot{\boldsymbol{p}}_0$ и $\dot{\boldsymbol{p}}_N$ могут быть известны, в противном случае их приближенно вычисляют по следующим формулам:

$$\dot{\boldsymbol{p}}_0 = \frac{\boldsymbol{p}_1 - \boldsymbol{p}_0}{|\boldsymbol{p}_1 - \boldsymbol{p}_0|},\ \dot{\boldsymbol{p}}_N = \frac{\boldsymbol{p}_N - \boldsymbol{p}_{N-1}}{|\boldsymbol{p}_N - \boldsymbol{p}_{N-1}|},$$

После решения системы $\boldsymbol{M} \cdot \dot{\boldsymbol{P}} = \boldsymbol{R}$ будут найдены все касательные векторы $\dot{\boldsymbol{p}}_i$ и можно вычислить коэффициенты для каждого сегмента кубического сплайна используя следующую формулу:

$$\boldsymbol{r}_i(\tau) = [F_{1i}(\tau), F_{2i}(\tau), F_{3i}(\tau), F_{4i}(\tau)] \begin{bmatrix} \boldsymbol{p}_i \\ \boldsymbol{p}_{i+1} \\ \dot{\boldsymbol{p}}_i \\ \dot{\boldsymbol{p}}_{i+1} \end{bmatrix}$$

которая получается из формулы (4.1) после перегруппировки элементов и записи в их в матричном виде. Параметр $\tau = t/t_{i+1}$, а функции $F(\tau)$ называются *весовыми* функциями и имеют следующий вид:

$$\begin{aligned} F_{1i}(\tau) &= 2\tau^3 - 3\tau^2 + 1, \\ F_{2i}(\tau) &= -2\tau^3 + 3\tau^2, \\ F_{3i}(\tau) &= \tau(\tau^2 - 2\tau + 1)t_{i+1}, \\ F_{4i}(\tau) &= \tau(\tau^2 - \tau)t_{i+1}. \end{aligned} \tag{4.4}$$

Так как в каждом i-ом сегменте при хордовой интерполяции параметр t изменяется в границах отрезка $[0, t_{i+1}]$, то $\tau = t/t_{i+1}$ изменяется в границах отрезка $[0, 1]$.

Таким образом, вычисление кубического сплайна при хордовой интерполяции включает в себя следующие шаги.

1. Задаются опорные точки $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_N$; значения касательных $\dot{\boldsymbol{p}}_0, \dot{\boldsymbol{p}}_1, \ldots, \dot{\boldsymbol{p}}_N$ считаются не известными и вычисляются позже.

2. По заданным опорным точкам вычисляются конечные точки t_i отрезков $[0, t_i]$ по формулам (4.2).

3. Для нахождения значений касательных векторов в опорных точках решается система линейных алгебраических уравнений (4.3).

4. По опорным точкам и касательным вычисляются коэффициенты каждого сегмента сплайна, для чего удобно использовать формулу (4.4).

5. Путем изменения значений параметра τ в каждом сегменте, производится интерполяция искомой кривой.

Кубические сплайны оказались особенно полезны для численных методов решения ОДУ с плотной выдачей, так как в случае ОДУ значения производных известны в каждой точке из правой части дифференциального уравнения и поэтому решать систему (4.3) нет необходимости.

4.3.5. Сплайн Эрмита

Сплайны Эрмита названы в честь французского математика Шарля Эрмита (Charles Hermite, 1822–1901) и являются кубическими полиномами, проходящими через две точки $\boldsymbol{p}_i$ и $\boldsymbol{p}_{i+1}$ при известных производных в этих точках $\boldsymbol{m}_i$ и $\boldsymbol{m}_{i+1}$. В компьютерной графике данные сплайны часто носят название `cspline`.

Для простоты обозначений рассмотрим две точки $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ и два касательных вектора $\boldsymbol{m}_0$, $\boldsymbol{m}_1$. Формула для вычисления точек сплайна имеет следующий вид:

$$\boldsymbol{r}(t) = h_{00}(t)\boldsymbol{p}_0 + h_{10}(t)\boldsymbol{m}_0 + h_{01}(t)\boldsymbol{p}_1 + h_{11}(t)\boldsymbol{m}_1,\ t \in [0,1].$$

Функции $h_{00}(t)$, $h_{01}(t)$, $h_{10}(t)$ и $h_{11}(t)$ являются базисными полиномами. Их можно вычислить воспользовавшись следующими граничными условиями:

$$\boldsymbol{r}(0) = \boldsymbol{p}_0,\ \boldsymbol{r}(1) = \boldsymbol{p}_1,\ \boldsymbol{r}'(0) = \boldsymbol{m}_0,\ \boldsymbol{r}'(1) = \boldsymbol{m}_1.$$

На практике используют три формы записи данных функций.

Приведенная форма совпадает с нормализованными весовыми функциями для кубических сплайнов:

$$h_{00}(t) = 2t^3 - 3t^2 + 1, \quad h_{10}(t) = t^3 - 2t^2 + t,$$
$$h_{01}(t) = -2t^3 + 3t^2, \qquad h_{11}(t) = t^3 - t^2.$$

Факторизиованная форма позволяет минимизировать число арифметических операций при вычислении:

$$h_{00}(t) = (1 + 2t)(1 - t)^2, \quad h_{10}(t) = t(1 - t)^2,$$
$$h_{01}(t) = t^2(3 - 2t), \qquad h_{11}(t) = t^2(t - 1).$$

Форма Безье показывает связь между сплайнами Эрмита и полиномами Бернштейна (см. далее)

$$h_{00}(t) = B_3^0(t) + B_3^1(t), \quad h_{10}(t) = B_3^1(t)/3,$$
$$h_{01}(t) = B_3^3(t) + B_3^2(t), \quad h_{11}(t) = -B_3^2(t)/3.$$

Факторизированная форма помогает сразу же выяснить, что $h_{00}(1) = h_{10}(1) = 0, h_{11}(1) = 0, h_{01}(1) = 1$ и $h_{01}(0) = h_{10}(0) = 0, h_{11}(0) = 0, h_{00}(0) = 1$.

Третья форма позволяет выявить связь сплайнов Эрмита с кривыми Безье и свести их построение к построению кривых Безье. Для этого можно вычислить опорные точки кривой Безье по следующим формулам:

$$\boldsymbol{w}_0 = \boldsymbol{p}_0,\ \boldsymbol{w}_1 = \boldsymbol{p}_0 + \boldsymbol{m}_0/3,\ \boldsymbol{w}_2 = \boldsymbol{p}_1 - \boldsymbol{m}_1/3,\ \boldsymbol{w}_3 = \boldsymbol{p}_1.$$

Сплайны Эрмита можно использовать для аппроксимации данных без известных значений производных (касательных). Для этого необходимо вычислить значения производных в точках по приближенным формулам.

Пусть дан набор точек $\{(t_k, \boldsymbol{p}_k)\}_1^n$ который необходимо соединить гладкой кривой. При использовании сплайнов Эрмита интерполяционная кривая будет состоять из отдельных сплайнов Эрмита, соединяющих пары точек. Кривая будет гладко дифференцируемой на все промежутке $[t_0, t_n]$. Приближенное вычисление касательных векторов (производных) можно проводить разными способами. Перечислим здесь несколько формул.

Простейший способ заключается в использовании трехточечных конечных разностей:

$$\boldsymbol{m}_k = \frac{1}{2}\left(\frac{\boldsymbol{p}_{k+1} - \boldsymbol{p}_k}{t_{k+1} - t_k} + \frac{\boldsymbol{p}_k - \boldsymbol{p}_{k-1}}{t_k - t_{k+1}}\right),\ k = 2, \dots, n-1.$$

В начальной и конечной точке придется использовать обычные конечные разности:

$$\boldsymbol{m}_0 = \frac{\boldsymbol{p}_1 - \boldsymbol{p}_0}{t_1 - t_0},\ \boldsymbol{m}_n = \frac{\boldsymbol{p}_n - \boldsymbol{p}_{n-1}}{t_n - t_{n-1}}.$$

Другим способом является использование формулы

$$\boldsymbol{m}_k = (1-c)\frac{\boldsymbol{p}_{k+1} - \boldsymbol{p}_{k-1}}{t_{k+1} - t_{k-1}},\ c \in [0,1].$$

При $c = 0$ сплайн Эрмита сведется к сплайну Катмулл–Рома.

Еще одна формула имеет вид:

$$\boldsymbol{m}_k = s_{k+1}\frac{\boldsymbol{p}_k - \boldsymbol{p}_{k-1}}{s_k + s_{k+1}} + s_k\frac{\boldsymbol{p}_{k+1} - \boldsymbol{p}_k}{s_k + s_{k+1}},\ s_k = |\boldsymbol{p}_k - \boldsymbol{p}_{k-1}|,\ s_{k+1} = |\boldsymbol{p}_{k+1} - \boldsymbol{p}_k|.$$

В конечных точках найдем $\boldsymbol{m}_0$ и $\boldsymbol{m}_n$ из условия равенства нулю третей производной радиус-вектора $\boldsymbol{r}'''(t) = 0$.

4.4. Кривые Безье

Кривые Безье названы в честь французского инженера компании Рено (Renault) Пьера Безье (Pierre Bézier, 1910–1999). Он опубликовал результаты своих исследований в 1962 году. Однако он был не первым, кто использовал данные кривые. К похожим результатам пришел другой французский инженер компании Ситроен (Citroën) — Поль де Кастельё (Paul de Casteljau, 1930) в 1959 году, но свои результаты он не опубликовывал. Кроме того, кривые Безье по своей сути являются полиномами Бернштейна, названными так в честь русского математика Сергея Натановича Бернштейна (1880–1968).

Кривые Безье нашли свое применения во многих областях инженерного дела, компьютерной графики и дизайна. Они активно используются в следующих типах программ:

- В системах автоматизированного проектирования (CAD и CAM), которые активно используются в инженерии, архитектуре и в машиностроении.
- В векторных графических редакторах, таких как Adobe Illustrator, CorelDRAW, Inkscape.
- В векторных форматах графических изображений, таких как SVG, OpenType fonts, PDF, PS и многих других.
- В специализированных языках программирования для построения векторных изображений: Tikz/PGF, Asymptote.

4.4.1. Основные определения

Базисом Бернштейна называется совокупность функций следующего вида:

$$B_n^i(t) = \mathrm{C}_n^i (1-t)^{n-i} t^i,\ t \in [0,1],$$

где индексы $i, n = 0, 1, 2, 3, ...$, а параметр t принимает значения из единичного промежутка. *Кривая Безье* является полиномом Бернштейна с векторными коэффициентами $\boldsymbol{w}_i$:

$$\boldsymbol{B}_n(t) = \sum_{i=0}^{n} B_n^i(t) \boldsymbol{w}_i,$$

где $\boldsymbol{w}_i = (x_i, y_i, z_i)$ в трехмерном случае, и $\boldsymbol{w}_i = (x_i, y_i)$ в двухмерном. Кривая проходит через начальную $\boldsymbol{w}_0$ и конечную $\boldsymbol{w}_n$ точки. Точки $\{\boldsymbol{w}_0, \boldsymbol{w}_1, \dots, \boldsymbol{w}_n\}$ формирует опорный «скелет» или «каркас» и называются *управляющими* или *контрольными* точками кривой Безье. Ломанная, образующаяся при последовательном соединении этих точек, называется *характеристической ломанной* кривой Безье. Часто управляющими точками называют лишь точки $\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_{n-1}$ лежащие вне кривой, отличая их таким образом от конечных точек $\boldsymbol{w}_0$ и $\boldsymbol{w}_n$ через которые проходит кривая.

Вычислим несколько первых функций Бернштейна

$$B_0^0 = 1,\ B_1^0 = 1 - t,\ B_1^1 = t,$$

для вычисления других функций можно воспользоваться рекуррентной формулой:

$$B_n^i = t B_{n-1}^{i-1} + (1-t) B_{n-1}^i,$$

которая доказывается путем подстановки выражения для B_{n-1}^{i-1} и B_{n-1}^i.

Базис Бернштена представляет собой разложение единицы. Это можно показать, воспользовавшись формулой для бинома Ньютона:

$$\sum_{i=0}^{n} B_n^i(t) = (t + (1-t))^n = 1^n = 1.$$

На практике используют кривые Безье 2, 3 и 4 порядков. При реализации кривых более высокого порядка, самым ресурсоёмким шагом является вычисление биноминальных коэффициентов C_n^i, поэтому разумным является вычислить эти коэффициентов заранее и сохранить их во вложенных списках.

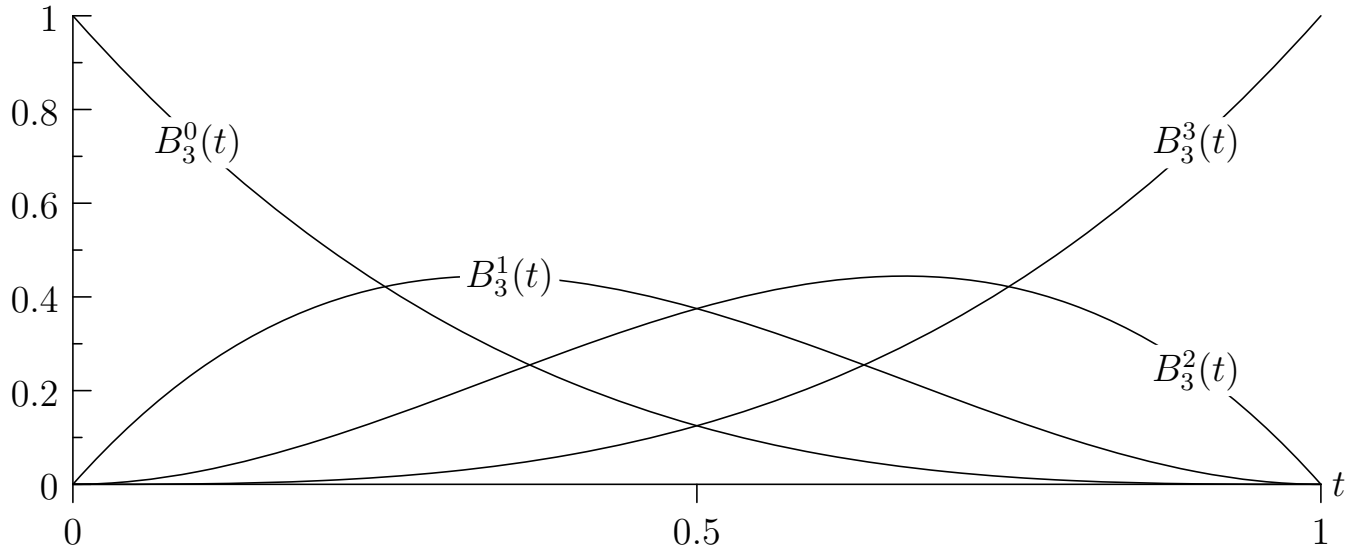

Рис. 4.3: Графики нескольких функций Бернштейна.

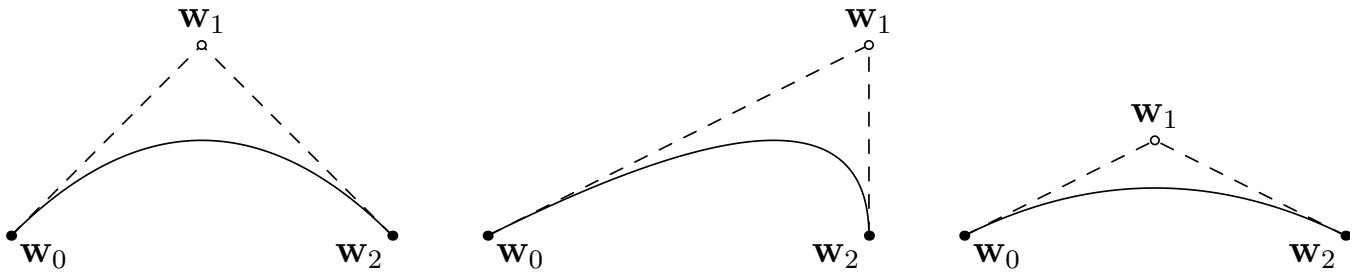

Рис. 4.4: Кривые Безье второго порядка. На рисунке показано различное положение контрольной точки $\boldsymbol{w}_1$.

На рисунке 4.4 изображены три кривые Безье второго порядка с одинаковыми начальной и конечной точками $\boldsymbol{w}_0$ и $\boldsymbol{w}_2$ и с разной контрольной точкой $\boldsymbol{w}_1$. Одной контрольной точки часто бывает недостаточно. Тогда используют кривые Безье третьего порядка (см. рис. 4.5) для построения которых нужны уже четыре опорные точки, две из которых являются контрольными. В графических векторных редакторах манипуляция кривыми Безье осуществляется путем перемещения опорных точек с помощью мыши. Вначале указываются желаемые конечные точки, а затем с помощью контрольных точек кривая подстраивается под нужную форму.

Выше упоминалось, что кривые Безье используются в шрифтах форматов True Type и Open Type. Каждый глиф шрифта состоит из одного или нескольких контуров, заданных с помощью кривых Безье третьего порядка с четырьмя опорными точками, две из которых управляющие. На рисунке 4.6 представлены несколько примеров.

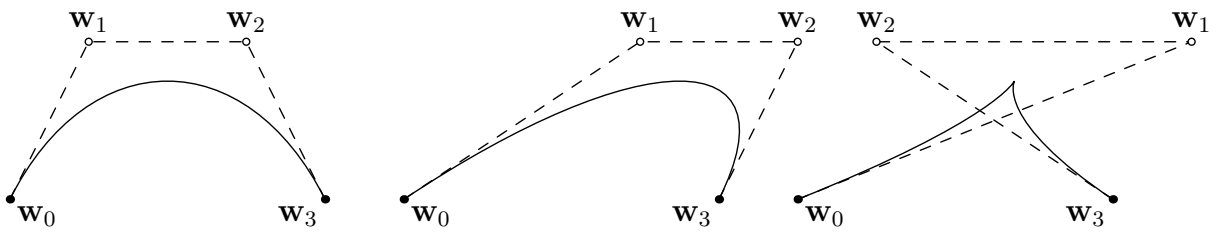

Рис. 4.5: Кривые Безье третьего порядка и их характеристические ломанные. Две контрольные точки $\boldsymbol{w}_1$ и $\boldsymbol{w}_2$ позволяют более гибко управлять положением кривой.

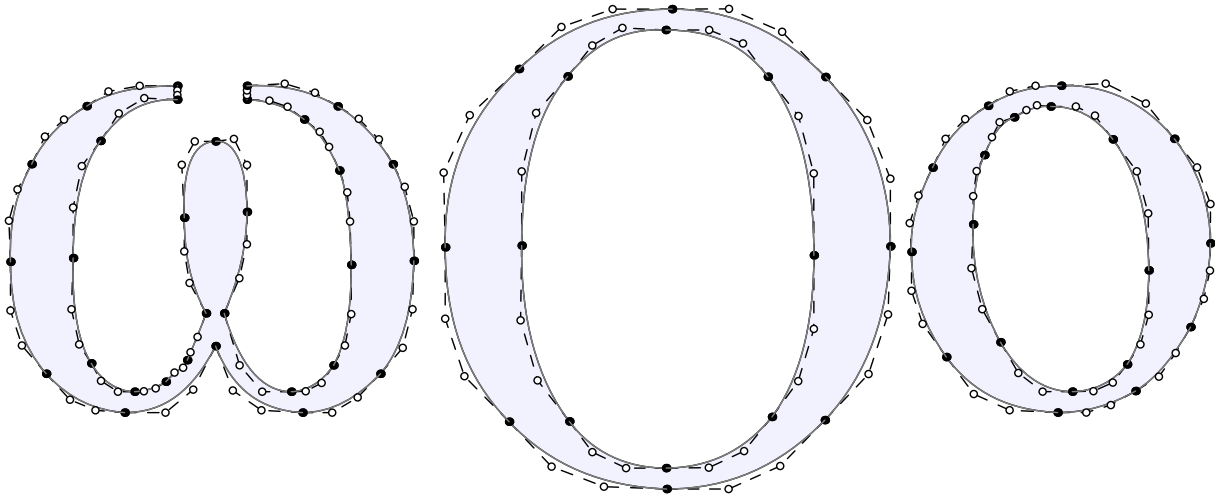

Рис. 4.6: Контуры букв ω, O и o шрифта Times New Roman. Показаны опорные и управляющие точки кривых Безье.

4.4.2. Матричные формулы для вычисления кривых Безье

Для ускорения вычислений точек кривой Безье можно воспользоваться матричной формулировкой вышеприведенных формул. Современные процессоры хорошо поддерживают операции с векторами, поэтом выигрыш в производительности модно получить даже при выполнении кода на одноядерном процессоре. В случае использовании библиотеки `NumPy` ускорение достигается как за счет вызовов быстрых функций на Си, так и за счет распараллеливания операций с массивами.

Мы рассмотрим случай кубической и квадратной кривых Безье. При необходимости можно аналогичным способом получить формулы и для кривых более высокого порядка.

Кубическая кривая Безье имеет следующий вид:

$$\boldsymbol{B}_3(t) = (1-t)^3\boldsymbol{w}_0 + 3(1-t)^2t\boldsymbol{w}_1 + 3(1-t)t^2\boldsymbol{w}_2 + t^3\boldsymbol{w}_3$$

Раскроем скобки в каждом слагаемом, однако подобные приводить не будем. Также временно уберем веса, заменив их на единицу и запишем слагаемые одно под другим, столбиком.

$$B_3(t) = \begin{pmatrix} 1 + 3t^2 - 3t - t^3 + \\ +3(t - 2t^2 + t^3) + \\ +3(t^2 - t^3) + \\ +t^3 \end{pmatrix} = \begin{pmatrix} -1t^3 & +3t^2 & -3t & +1 & + \\ +3t^3 & -6t^2 & +3t & +0 & + \\ -3t^3 & +3t^2 & +0t & +0 & + \\ +1t^3 & +0t^2 & +0t & +0 & \end{pmatrix}$$

Разложим теперь данную сумму на произведения вектора-строки $(t^3, t^2, t^1, 1)$ и векторов-столбцов, составленных из коэффициентов:

$$B_3(t) = \begin{pmatrix} t^3 & t^2 & t^1 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ 3 \\ -3 \\ 1 \end{pmatrix} + \begin{pmatrix} t^3 & t^2 & t^1 & 1 \end{pmatrix} \begin{pmatrix} -3 \\ -6 \\ 3 \\ 0 \end{pmatrix} +$$

$$+ \begin{pmatrix} t^3 & t^2 & t^1 & 1 \end{pmatrix} \begin{pmatrix} -3 \\ -3 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} t^3 & t^2 & t^1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\boldsymbol{B}_3(t) = \begin{pmatrix} t^3 & t^2 & t^1 & 1 \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & t^1 & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

Возвращаем опорные точки

$$\boldsymbol{B}_3(t) = \begin{pmatrix} 1 & t^1 & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{w}_0 \\ \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \boldsymbol{w}_3 \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & t^1 & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{pmatrix}$$

В результате получим

$$\boldsymbol{B}_3(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

Кривая Безье второго порядка задается следующей формулой:

$$\boldsymbol{B}_2(t) = (1-t)^2 \boldsymbol{w}_0 + 2t(1-t)\boldsymbol{w}_1 + t^2 \boldsymbol{w}_2,$$

и в матричном виде представима следующим образом:

$$\boldsymbol{B}_2(t) = \begin{pmatrix} 1 & t^1 & t^2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ -2 & 2 & 0 \\ 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}$$

4.4.3. Алгоритм де Кастельё

Можно сформулировать итерационную процедуру, которая позволит вычислять точки кривой Безье без использования функций Бернштейна. Геометрический смысл данной процедуры заключается в построении дополнительного каркаса из ломанных линий.

Выше было показано, что функции Бернштейна удовлетворяют следующему рекуррентному соотношению:

$$B_n^i(t) = tB_{n-1}^{i-1}(t) + (1-t)B_{n-1}^i(t).$$

Используя это соотношение можно преобразовать формулу для кривых Безье:

$$\boldsymbol{r}(t) = \sum_{i=0}^{n} B_n^i(t)\boldsymbol{p}_i = B_n^0(t)\boldsymbol{p}_0 + B_n^n(t)\boldsymbol{p}_n + \sum_{i=1}^{n-1} B_n^i(t)\boldsymbol{p}_i,$$

учитывая, что $B_n^0(t) = (1-t)^n$ и $B_n^n(t) = t^n$ получим

$$\begin{aligned}\boldsymbol{r}(t) &= (1-t)^n\boldsymbol{p}_0 + t^n\boldsymbol{p}_n + \sum_{i=1}^{n-1}(tB_{n-1}^{i-1}(t) + (1-t)B_{n-1}^i(t))\boldsymbol{p}_i = \\ &= (1-t)^n\boldsymbol{p}_0 + \sum_{i=1}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_i + t^n\boldsymbol{p}_n + \sum_{i=1}^{n-1} tB_{n-1}^{i-1}(t)\boldsymbol{p}_i = \\ &= (1-t)\Big[(1-t)^{n-1}\boldsymbol{p}_0 + \sum_{i=1}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_i\Big] + t\Big[t^{n-1}\boldsymbol{p}_n + \sum_{i=1}^{n-1} B_{n-1}^{i-1}(t)\boldsymbol{p}_i\Big]\end{aligned}$$

Для дальнейшего преобразования заметим, что $B_{n-1}^0(t) = (1-t)^{n-1}$ из чего следует, что $(1-t)^{n-1}\boldsymbol{p}_0 = B_{n-1}^0(t)\boldsymbol{p}_0$ и $B_{n-1}^{n-1}(t) = t^{n-1}$ из чего в свою очередь следует, что $t^{n-1}\boldsymbol{p}_n = B_{n-1}^0\boldsymbol{p}_n$. Используя эти соотношения продолжим преобразовывать формулу для кривой Безье.

$$\begin{aligned}(1-t)\Big[(1-t)^{n-1}\boldsymbol{p}_0 + \sum_{i=1}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_i\Big] + t\Big[t^{n-1}\boldsymbol{p}_n + \sum_{i=1}^{n-1} B_{n-1}^{i-1}(t)\boldsymbol{p}_i\Big] &= (1-t)\Big(\sum_{i=0}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_i\Big) + \\ &+ t\Big(\sum_{i=0}^{n-1} B_{n-1}^i\boldsymbol{p}_{i+1}\Big) = (1-t)\boldsymbol{r}_{n-1}^0(t) + t\boldsymbol{r}_{n-1}^1(t).\end{aligned}$$

Мы ввели следующие обозначения

$$\boldsymbol{r}_{n-1}^0(t) = \sum_{i=0}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_i, \quad \boldsymbol{r}_{n-1}^1 = \sum_{i=0}^{n-1} B_{n-1}^i(t)\boldsymbol{p}_{i+1}.$$

Продолжая процесс разложения дальше приходим к равенствам

$$\boldsymbol{r}_0^i = \sum_{j=0}^{0} B_0^0(t)\boldsymbol{p}_i = \boldsymbol{p}_i, i = 0, 1, \dots, n$$

$$\boldsymbol{r}_n^0(t) = \boldsymbol{r}(t), \quad \boldsymbol{r}_0^i = \boldsymbol{p}_i,$$

Получили рекуррентную формулу, которая позволяет в результате итерационного процесса вычислить координаты точек кривой Безье используя в качестве начальных данных лишь опорные точки $\boldsymbol{p}_i$. При этом пропадает необходимость вычислять функции Бернштейна.

$$\boldsymbol{r}_k^i = (1-t)\boldsymbol{r}_{k-1}^i + t\boldsymbol{r}_{k-1}^{i+1}, \; i + k \leqslant n$$

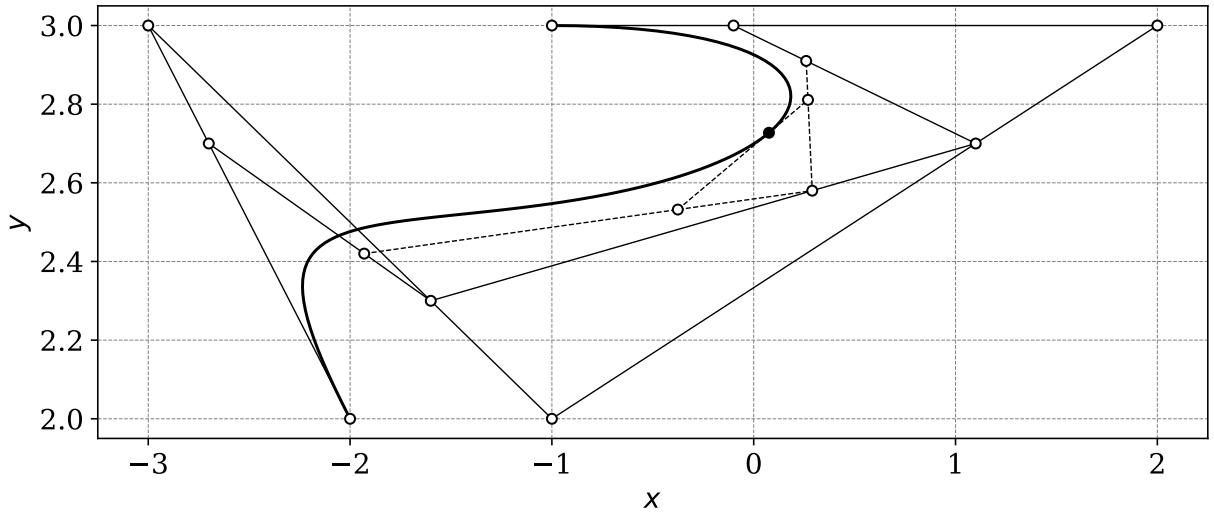

Рис. 4.7: Вспомогательные характеристические (опорные) ломанные, точки которых вычисляет алгоритм де Кастельё.

Данная формула называется *алгоритмом де Кастельё*. Геометрический смысл этого алгоритма заключается в вычислении вспомогательных характеристических ломанных, которые строятся друг за другом.

Чтобы лучше понять этот алгоритм, рассмотрим вычисление для кривой Безье 3-его порядка. Начинаем с 4-х опорных точек $\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$ и с их помощью вычисляем три новых опорных точки.

$$\begin{aligned}
\boldsymbol{r}_1^0 &= (1-t)\boldsymbol{r}_0^0 + t\boldsymbol{r}_0^1 = (1-t)\boldsymbol{p}_0 + t\boldsymbol{p}_1, \\
\boldsymbol{r}_1^1 &= (1-t)\boldsymbol{r}_0^1 + t\boldsymbol{r}_0^2 = (1-t)\boldsymbol{p}_1 + t\boldsymbol{p}_2, \\
\boldsymbol{r}_1^2 &= (1-t)\boldsymbol{r}_0^2 + t\boldsymbol{r}_0^3 = (1-t)\boldsymbol{p}_2 + t\boldsymbol{p}_3.
\end{aligned}$$

Вычисленные три точки $\boldsymbol{r}_1^0, \boldsymbol{r}_1^1, \boldsymbol{r}_1^2$ используем для вычисления следующих двух:

$$\begin{aligned}
\boldsymbol{r}_2^0 &= (1-t)\boldsymbol{r}_1^0 + t\boldsymbol{r}_1^1, \\
\boldsymbol{r}_2^1 &= (1-t)\boldsymbol{r}_1^1 + t\boldsymbol{r}_2^1.
\end{aligned}$$

Наконец последние две точки $\boldsymbol{r}_2^0, \boldsymbol{r}_2^1$ дают нам возможность вычислить точку непосредственно на самой кривой Безье:

$$\boldsymbol{r}_3^0 = (1-t)\boldsymbol{r}_2^0 + t\boldsymbol{r}_2^1.$$

4.4.4. Нахождение производных от кривых Безье

Одним из полезных свойств кривой Безье является возможность вычислить значение производной в произвольной точке лишь пересчитав опорные точки. Пусть заданы опорные точки $\boldsymbol{w}_0, \boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ и по ним построена кривая Безье:

$$\boldsymbol{r}(t) = \sum_{i=0}^{n} B_n^i(t)\boldsymbol{w}_i.$$

Можно доказать следующую формулу:

$$\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} = \sum_{i=0}^{n-1} B_{n-1}^{i}(t) n(\boldsymbol{w}_{i+1} - \boldsymbol{w}_i),$$

где $n(\boldsymbol{w}_{i+1} - \boldsymbol{w}_i)$ — новый набор опорных точек для кривой $n-1$-го порядка, которой и является производная от исходной кривой Безье. Для доказательства этой формулы необходимо вычислить производные от функций Бенштейна и перегруппировать их сумму.

Приведем простейший пример для кривой третьего порядка. Пусть даны четыре опорные точки $\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3$, тогда опорные точки, соответствующие первой производной будут иметь вид: $\{\boldsymbol{w}_0', \boldsymbol{w}_1', \boldsymbol{w}_2'\} = \{3(\boldsymbol{w}_1 - \boldsymbol{w}_0), 3(\boldsymbol{w}_2 - \boldsymbol{w}_1), 3(\boldsymbol{w}_3 - \boldsymbol{w}_2)\}$. Опорные точки для второй приводной можно вычислить аналогично $\{\boldsymbol{w}_0'', \boldsymbol{w}_1''\} = \{2(\boldsymbol{w}_1' - \boldsymbol{w}_0'), 2(\boldsymbol{w}_2' - \boldsymbol{w}_1')\}$. Третья производная представляет собой только одну точку.

Возможность быстро вычислять производные всех возможных порядков позволяет эффективно находить касательные и нормальные векторы к кривой, а также находить экстремальные точки. Рассмотрим формулы для вычисления касательных и нормальных векторов.

Компоненты ненормированного касательного вектора находятся сразу же после вычисления производной, так как

$$\boldsymbol{v}(t) = \frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} = \left(\frac{\mathrm{d}x(t)}{\mathrm{d}t}, \frac{\mathrm{d}y(t)}{\mathrm{d}t} \right)^T, \quad |\boldsymbol{v}(t)| = \sqrt{\dot{x}^2(t) + \dot{y}^2(t)}.$$

Нормируем вектор

$$\boldsymbol{v}_0(t) = \frac{\boldsymbol{v}(t)}{|\boldsymbol{v}|} = \left(\frac{\dot{x}(t)}{|\boldsymbol{v}(t)|}, \frac{\dot{y}(t)}{|\boldsymbol{v}(t)|} \right)^T = (v_0^1, v_0^2)^T.$$

Задачу нахождения нормали можно существенно упростить, если воспользоваться тем фактом, что нормаль и касательный вектор ортогональны друг-другу, так что нормаль можно получить из касательного вектора путем поворота последнего на 90°

$$\boldsymbol{n}_0 = \begin{pmatrix} v_0^1 \cos \pi/2 - v_0^2 \sin \pi/2 \\ v_0^1 \sin \pi/2 + v_0^2 \cos \pi/2 \end{pmatrix} = \begin{pmatrix} -v_0^2(t) \\ +v_0^1(t) \end{pmatrix} = \begin{pmatrix} \dfrac{-\dot{y}}{\sqrt{\dot{x} + \dot{y}}} \\ \dfrac{+\dot{x}}{\sqrt{\dot{x} + \dot{y}}} \end{pmatrix}$$

Таким образом:

$$\boldsymbol{v}_0(t) = \begin{pmatrix} \dfrac{\dot{x}}{\sqrt{\dot{x} + \dot{y}}} \\ \dfrac{\dot{y}}{\sqrt{\dot{x} + \dot{y}}} \end{pmatrix}, \quad \boldsymbol{n}_0(t) = \begin{pmatrix} \dfrac{-\dot{y}}{\sqrt{\dot{x} + \dot{y}}} \\ \dfrac{\dot{x}}{\sqrt{\dot{x} + \dot{y}}} \end{pmatrix},$$

4.5. Сплайн Катмулла–Рома

Сплайны Катмулла–Рома — это семейство кубических интерполяционных сплайнов, отличающихся той особенностью, что касательная в каждой точке вычисляется с использованием предыдущей и последующей точек. Сплайны названы в честь Эдвина Катмулла

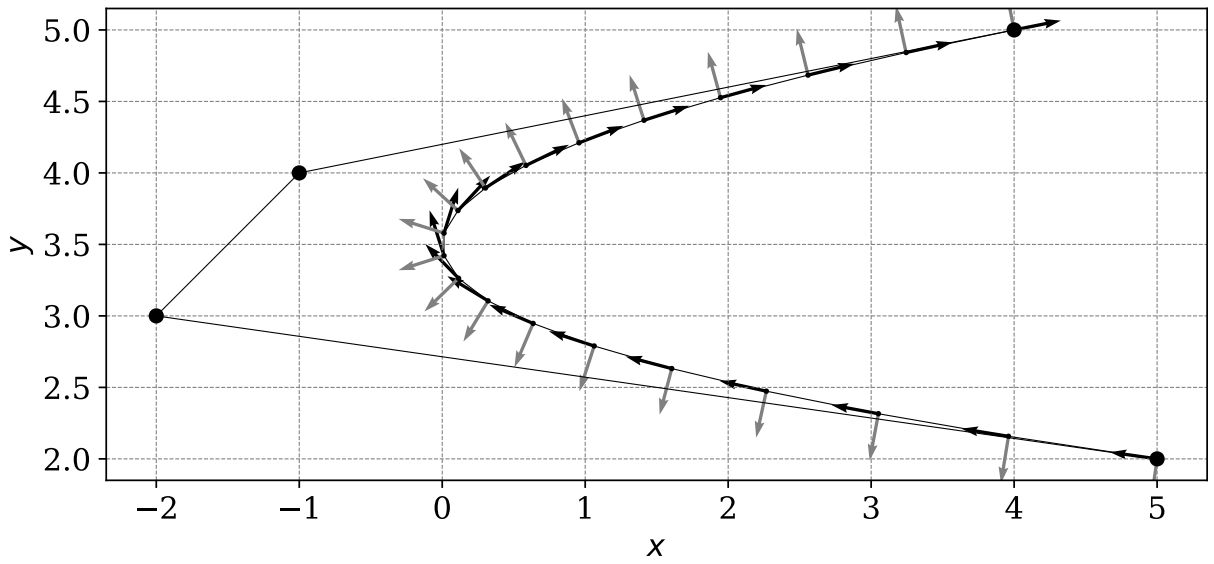

Рис. 4.8: Касательные и нормали к точкам кривой Безье третьего порядка.

(Edwin Catmull, 1945) и Рафаэля Рома (Raphael Rom). Эдвин Катмулл по образованию компьютерный инженер, работал в студии спецэффектов Industrial Light and Magic (основатель Джордж Лукас), в Walt Disney Animation Studios и Pixar. В настоящее время является президентом последних двух анимационных студий.

Для вычисления сплайнов следует воспользоваться следующей матричной формулой:

$$\boldsymbol{r}(t) = \begin{pmatrix} 1 & t & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_{i-2} \\ \boldsymbol{p}_{i-1} \\ \boldsymbol{p}_i \\ \boldsymbol{p}_{i+1} \end{pmatrix}$$

Сплайн Катмулла–Рома имеет непрерывную производную во всех точках, но лежит вовне своей характеристической ломанной. Параметр τ называется параметром натяжения и от него зависит насколько резко кривая огибает контуры опорных точек.

Важно отметить, что из четырех опорных точек $\boldsymbol{p}_{i-2}$, $\boldsymbol{p}_{i-1}$, $\boldsymbol{p}_i$ и $\boldsymbol{p}_{i+1}$ первая и последняя игнорируются и кривая строится только от $\boldsymbol{p}_{i-1}$ до $\boldsymbol{p}_i$.

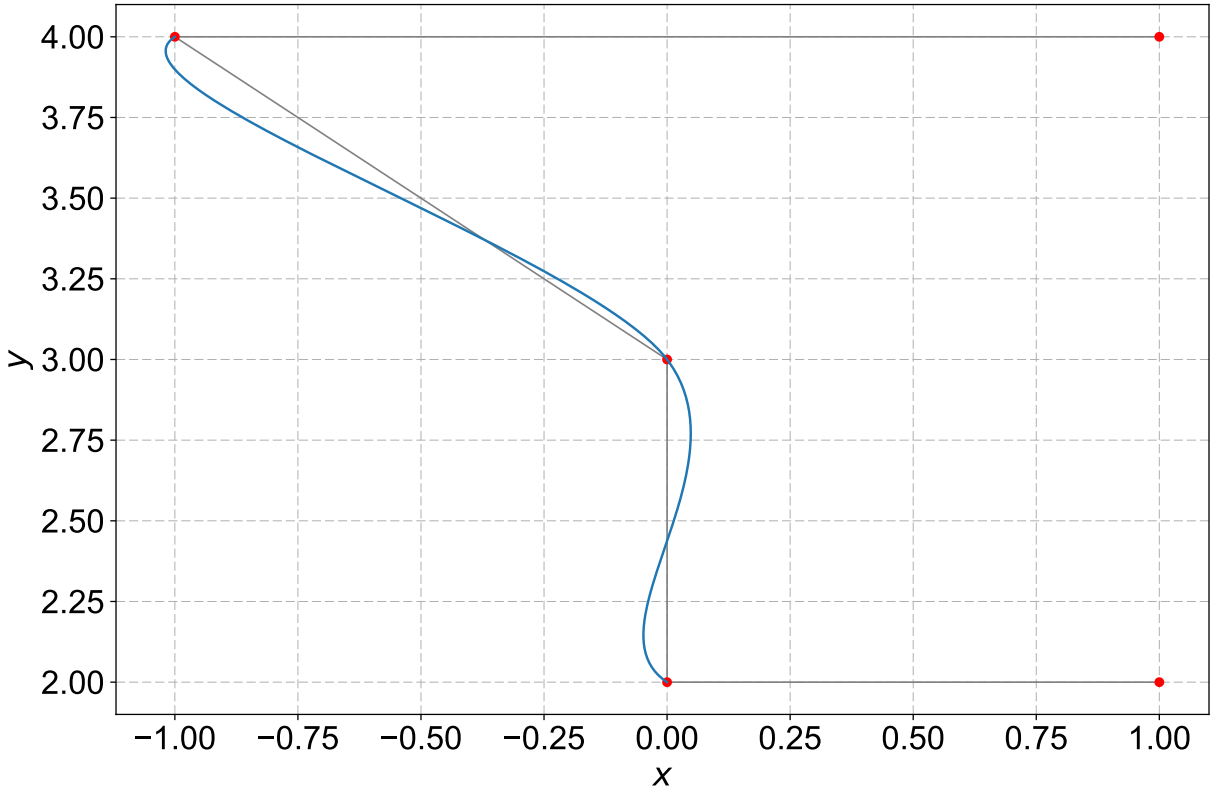

Рис. 4.9: Сплайн Катмулла–Рома для 5 опорных точек.

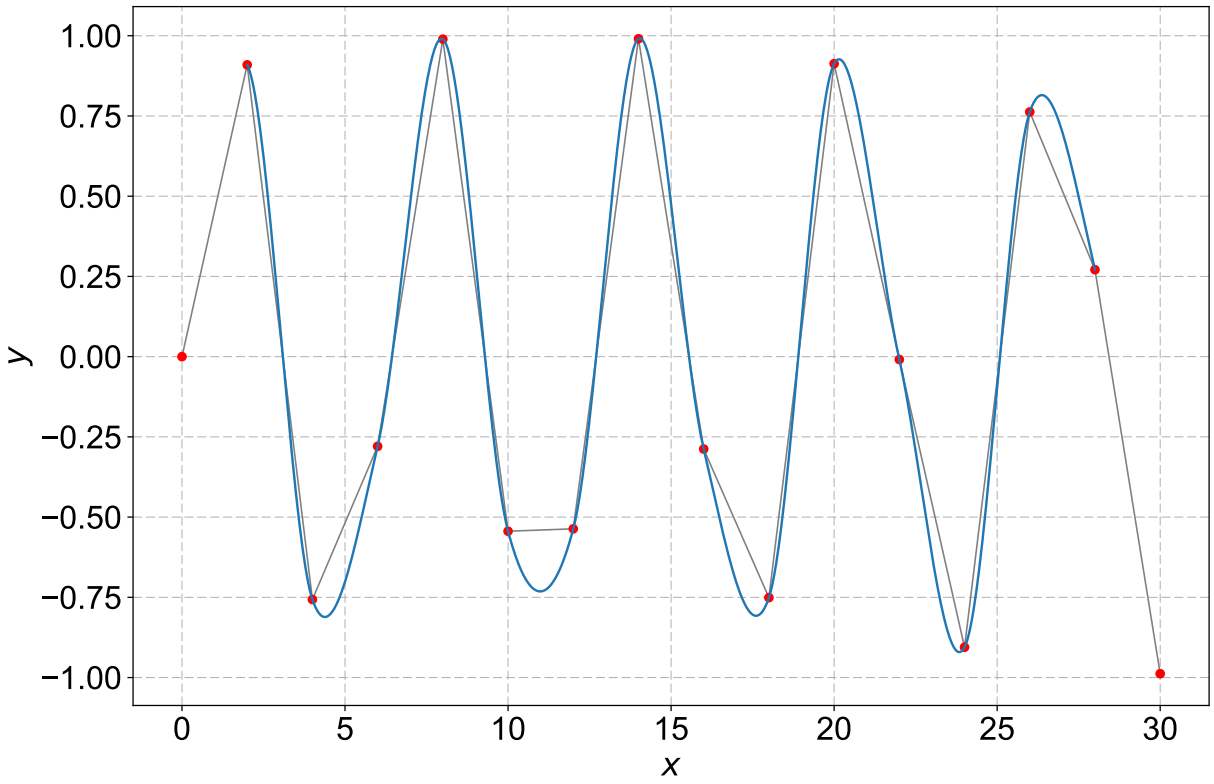

Рис. 4.10: Интерполяции функции sin с помощью сплайнов Катмулла–Рома.

4.6. Вопросы для самопроверки

1. Что изучает компьютерная геометрия? Чем она отличается от компьютерной (машинной) графики?
2. Математический аппарат каких теоретических разделов математики используется в компьютерной геометрии?
3. Дайте определение плоской и трехмерной кривой, которое используется в компьютерной геометрии.
4. Какие три основных способа записи уравнений, задающих кривую на плоскости и в пространстве? Какой из них наиболее часто используется при построении кривых с помощью компьютера?
5. Что такое натуральный параметр кривой? Как его найти? Каков его геометрический смысл?
6. Какие векторы составляют репер Френе в двумерном и трехмерном случае?
7. Что такое кривизна кривой? Что такое центр кривизны? Радиус кривизны?
8. Что такое резольвента и эволюта? Как они связаны?
9. Что такое сплайн и чем он отличается от полинома?
10. Какие сложности могут проявится при интерполяции полиномами высокого порядка? Как этих сложностей избежать?
11. Запишите параметрическое уравнение для ломанной линии.
12. Запишите формулы для кубического сплайна. Чем кубический сплайн отличается от кубического полинома?
13. Запишите формулы для сплайнов Ньютона и Лагранжа. Чем данные сплайны отличаются от одноименных полиномов?
14. Запишите формулы для сплайна Эрмита. Какие формы записи его коэффициентов вы знаете?
15. Каковы преимущества и недостатки сплайнов Эрмита по сравнению со сплайнами Ньютона, Лагранжа и кубическими сплайнами?
16. Чем сплайн Эрмита отличается от одноименных полиномов?
17. Дайте определение кривым Безье. В каких областях они применяются?
18. Что такое функции Бернштейна и как они связаны с кривыми Безье?
19. Что такое опорные точки кривых Безье?

20. Запишите матричные формулы для вычисления кривых Безье второго и третьего порядков. Какие преимущество дает такая форма записи?

21. Сформулируйте алгоритм де Кастельё. Какие преимущества он дает при построении кривой Безье?

22. Как построить опорные ломанные для кривой Безье?

23. Как наиболее просто найти производные для каждой точки кривой Безье?

24. Как сплайны Эрмита связаны со сплайнами Безье?

25. Дайте определение сплайну Катмулла–Рома. Какие особенности отличают его от ранее изученных сплайнов?

26. Как строится опорная ломанная для сплайнов Катмулла–Рома?

27. Как сплайны Катмулла–Рома связаны со сплайнами Эрмита?

28. Запишите формулы для сплайна Катмулла–Рома четвертого порядка.

Список литературы

1. *Любанович Б.* Простой Python. Современный стиль программирования. — Москва : Питер, 2017. — ISBN 978-5-496-02088-6.
2. *Слаткин Б.* Секреты Python : 59 рекомендаций по написанию эффективного кода. — Москва : Вильямс, 2016. — ISBN 9785845920782.
3. *Маккинни У.* Python и анализ данных. — Москва : ДМК Пресс, 2015. — ISBN 978-5-97060-315-4.
4. *Плас Д. В.* Python для сложных задач. Наука о данных и машинное обучение. — Москва : Питер, 2020. — ISBN 978-5-496-03068-7.
5. Python home site. — 2020. — URL: `https://www.python.org/`.
6. *Rossum G.* Python Reference Manual : тех. отч. — Amsterdam, The Netherlands, The Netherlands, 1995.
7. NumPy home site. — 2020. — URL: `http://www.numpy.org//`.
8. *Jones E.*, *Oliphant T.*, *Peterson P.* [и др.]. SciPy: Open source scientific tools for Python. — 2001–. — URL: `http://www.scipy.org/`.
9. SymPy home site. — 2020. — URL: `http://www.sympy.org/ru/index.html`.
10. Matplotlib home site. — 2020. — URL: `https://matplotlib.org/`.
11. Numba home site. — 2020. — URL: `https://numba.pydata.org/`.
12. MatLab home site. — 2020. — URL: `https://www.mathworks.com/`.
13. Mathematica home site. — 2020. — URL: `https://www.wolfram.com/mathematica/`.
14. *B. Monagan M.*, *O. Geddes K.*, *Heal K. M.*, *Labahn G.*, *M. Vorkoetter S.*, *McCarron J.*, *DeMarco P.* Maple 10 Programming Guide. — Waterloo ON, Canada : Maplesoft, 2005.
15. Maple home site. — 2020. — URL: `https://www.maplesoft.com/products/maple/`.
16. Cpython git repository. — 2020. — URL: `https://github.com/python/cpython`.
17. Jython home site. — 2020. — URL: `http://www.jython.org/`.
18. PyPy home site. — 2020. — URL: `https://pypy.org/`.
19. IronPython home site. — 2020. — URL: `http://ironpython.net/`.
20. *Голованов Н. Н.* Геометрическое моделирование. — Москва : Физматлит, 2002. — ISBN 5940520480.

21. *Роджерс Д.*, *Адамс А.* Математические основы машинной графики. — Москва : Мир, 2001. — ISBN 5030021434.

22. *А. Н. Е.* Компьютерная геометрия и алгоритмы машинной графики. — Санкт-Петербург : БХВ-Петербург, 2003.

23. *Мищенко А.*, *Фоменко А.* Краткий курс дифференциальной геометрии и топологии. — Москва : ФИЗМАТЛИТ, 2017. — ISBN 978-5-9710-2681-5.

24. *Шаров Г. С.*, *Шелехов А. М.*, *Шестакова М. А.* Дифференциальная геометрия и топология в задачах. — Москва : Ленанд, 2017. — ISBN 978-5-9710-3743-9.

25. *Щепетилов А. В.* Введение в дифференциальную геометрию. — Москва : КДУ, 2017. — ISBN 978-5-91304-710-6.

26. *Рашевский П.* Риманова геометрия и тензорный анализ. 1: Евклидовы пространства и аффинные пространства. Тензорный анализ. Математические основы специальной теории относительности. — Москва : УРСС, 2014. — ISBN 978-5-396-00577-8.

27. *Рашевский П.* Риманова геометрия и тензорный анализ. 2: Римановы пространства и пространства аффинной связности. Тензорный анализ. Математические основы общей теории относительности. — Москва : УРСС, 2014. — ISBN 978-5-396-00578-5.

28. *Фиников С.* Курс дифференциальной геометрии. — Москва : URSS, 2017.

29. *Дубровин Б.*, *Новиков С.*, *Фоменко А.* Современная геометрия: Методы и приложения. 1: Геометрия поверхностей, групп преобразований и полей. — 6-е изд. — Москва : УРСС: Книжный дом «ЛИБРОКОМ», 2013. — ISBN 978-5-453-00047-0.

30. *Дубровин Б.*, *Новиков С.*, *Фоменко А.* Современная геометрия: Методы и приложения. 2: Геометрия и топология многообразий. — 6-е изд. — Москва : УРСС: Книжный дом «ЛИБРОКОМ», 2013. — ISBN 978-5-453-00048-7.