МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Новосибирский национальный исследовательский государственный университет»
(Новосибирский государственный университет, НГУ)
Структурное подразделение Новосибирского государственного университета
Высший колледж информатики Университета (ВКИ НГУ)
КАФЕДРА ЕСТЕСТВЕННОНАУЧНЫХ ДИСЦИПЛИН

Цифровая обработка сигналов и быстрых методов дискретного анализа на языке программирования Python.

УЧЕБНОЕ ПОСОБИЕ

Новосибирск
2020

УДК 004.67, 621.372

Рекомендовано координационным методическим советом ВКИ НГУ
в качестве учебного пособия для студентов

Рецензент

Токарев Михаил Петрович, к.ф.-м.н., старший научный сотрудник федерального государственного бюджетного учреждения науки института теплофизики им. С.С. Кутателадзе.

И.А. Козулин

Цифровая обработка сигналов и быстрых методов дискретного анализа на языке программирования Python. Учебное пособие / И.А. Козулин, Е.А. Меркулова. - Новосибирск: Структурное подразделение Новосибирского государственного университета Высший колледж информатики (ВКИ НГУ), 2020. –82 с.

Учебное пособие направлено на развитие практических навыков и закреплению теоретического материала, полученного в рамках курса “Цифровая обработка сигналов”. Методическое пособие поможет читателю лучше понять основы цифровой обработки сигналов и дополнительно освоить тонкости программирования на языке Python. В пособие представлены теоретические основы современных методов цифровой обработки сигналов и важнейших алгоритмов, применяемых при компьютерной обработки результатов физического эксперимента.

Учебное пособие адресовано студентам высших учебных заведений, обучающимся по направлению подготовки “Мехатроника и робототехника”.

Предисловие

Цифровая обработка сигналов – наука о представлении сигналов в цифровом виде методах обработки таких сигналов. Она охватывает множество предметных областей, таких как обработка изображений и биомедицинских данных, обработка звука и речи, обработка сигналов с сонаров, радаров и сенсоров, спектральный анализ. Данное методическое пособие направлено на изучение прикладных аспектов цифровой обработки сигналов, которую удобно реализовать на языке программирования Python. Данный язык достаточно быстро развивается и является одним из самых популярных и адаптивных. Это высокоуровневый язык программирования общего назначения. Синтаксис языка минималистичен, что позволяет увеличить производительность разработчика и читаемость кода.

Пособие направлено на закрепление теоретических основ современных методов и важнейших алгоритмов, применяемых при компьютерной обработке результатов физического эксперимента. Можно сказать, что дисциплина «Цифровая обработка сигналов» является базовой для освоения таких дисциплин как техническое зрение, робототехника, модели и методы искусственного интеллекта.

Текущий контроль по дисциплине проводится в течение всего семестра. Студенты в течение семестра должны решить задачи, изучить дополнительный материал, необходимый для выполнения задания (задач) в соответствии с календарным графиком.

Правила оформления лабораторных работ:

−лабораторные работы выполняются на языке Python;

−программа должна сопровождаться кратким описанием в виде комментария в заголовке программы, если она несложная, или отдельного текста, описывающего, что делает эта программа, какой алгоритм использует и как вызывается, с какими параметрами. Каждая функция должна предваряться комментарием с информацией о назначении функции, типах параметров и возвращаемого значения и с кратким описанием алгоритма, если он нетривиален.

ПРАКТИКА ПРОГРАММИРОВАНИЯ НА PYTHON

1.1. Введение в программирование на Python.

Python - это интерпретируемый объектно-ориентированный язык программирования высокого уровня с динамической семантикой. Встроенные высокоуровневые структуры данных в сочетании с динамической типизацией и связыванием делают язык привлекательным для быстрой разработки приложений (RAD, Rapid Application Development). Синтаксис ядра Python минималистичен и прост в изучении, что позволит быстро его применять для обработки цифровых сигналов. У него есть свои преимущества и недостатки, а также область применения. Python поставляется с обширной стандартной библиотекой для широкого спектра задач. Кроме того, в Python есть довольно простые инструменты для интеграции с языками C, C ++ (и Java). Python поддерживает модули и использует несколько парадигм программирования: императивный (процедурный, структурный, модульный подходы), объектно-ориентированное и функциональное программирование. Интерпретатор Python и большая стандартная библиотека доступны бесплатно как исходные для всех основных платформ и могут свободно распространяться. В то же время стандартная библиотека включает большой объём полезных функций. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений, высокоуровневые структуры данных. Поддерживается разбиение программ на модули, которые, в свою очередь, могут объединяться в пакеты.

Создание Python было начато Гвидо ван Россумом (Guido van Rossum) в 1991 году, когда он работал над распределенной ОС Амеба. Ему требовался расширяемый язык, который бы обеспечил поддержку системных вызовов. За основу были взяты ABC и Модула-3. Название языка произошло вовсе не от названия семейства пресмыкающихся. Автор назвал язык в честь популярного британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона» [1]. Разработчики языка Python придерживаются определённой философии программирования, называемой «The Zen of Python» («Дзен Пито́на», или «Дзен Па́йтона») [2]. Её текст выдаётся интерпретатором Python по команде:

import this.

Автором этой философии считается Тим Петерс:

Текст философии:

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если не замалчиваются явно.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один — и, желательно, только один — очевидный способ сделать это.
- Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда.

- Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имён — отличная вещь! Давайте будем делать их больше!

Наличие дружелюбного, отзывчивого сообщества пользователей считается наряду с дизайнерской интуицией Гвидо одним из факторов успеха Python. Развитие языка происходит согласно чётко регламентированному процессу создания, обсуждения, отбора и реализации документов PEP (англ. Python Enhancement Proposal) — предложений по развитию Python и активно совершенствуется в настоящее время.

В Python последовательные действия описываются последовательными строками программы, следует отметить, что отступ в программе важен, все операторы в последовательности действий должны иметь одинаковые отступы:

```
a = 8
b = 1
a = a - b
b = a + b
c = a * b
print (a, b, c)

# комментарии в языке Python
# оператор условия и выбора:
if a < b:
   b = c
else:
   c = b
print (a, b, c)
```

Более общий случай оператора условия if:

сокращенный else if	**вложенный оператор ветвления**

`if a < 0:`
`  b = -2`
`elif a == 0:`
`  b = 4`
`else:`
`  b = 9`
	`if a < 0:`
	`  b = -2`
	`else:`
	`  if a == 0:`
	`  b = 4`
	`  else:`
	`  b = 9`

С помощью цикла можно описать повторяющиеся действия. В Python имеются два вида циклов: цикл for (пока выполняется некоторое условие) и цикл while (для всех значений последовательности):

```
s = "abcdefghijklmnoprst"
while s != "":
   print (s)
   s = s[1:-1]
```

Следующий пример читает строки из файла и выводит те, у которых длина больше 5:

```
f = open("file.txt", "r")
while 1:
   l = f.readline()
   if not l:
      break
   if len(l) > 5:
      print (l),
f.close()
```

Цикл for выполняет тело цикла для каждого элемента последовательности. В следующем примере выводится таблица умножения:

```
for i in range(1, 10):
   for j in range(1, 10):
      print ("%2i" % (i*j)),
   print ()
```

В Python можно определять собственные функции двумя способами: с помощью оператора **def** или прямо в выражении, посредством **lambda**. Второй способ будет рассмотрен позже, Пример определения и вызова функции:

```
def  position(x, y=0):
   return "X=%i, Y=%i" % (x, y)

print (position (3, 4.5))
print (position (10.5))
print (position (x=3, y=5))
```

Как уже говорилось, все данные в Python представлены объектами. Имена являются лишь ссылками на эти объекты и не несут нагрузки по декларации типа. Значения встроенных типов имеют специальную поддержку в синтаксисе языка: можно записать литерал строки, числа, списка, кортежа, словаря (и их разновидностей). Синтаксическую же поддержку операций над встроенными типами можно легко сделать доступной и для объектов определяемых пользователями классов.

1.1.1. Встроенные типы данных и классификация коллекций

Тип переменных **int** (целые числа) и **long** (целые произвольной точности) служат моделью для представления целых чисел. Значения типа int должны покрывать диапазон от -2147483648 до 2147483647, а точность целых произвольной точности зависит от объема доступной памяти, [3].

Тип **float** соответствует С-типу double для используемой архитектуры. Записывается традиционным способом либо через точку, либо в нотации с экспонентой:

Тип **complex**. Литерал мнимой части задается добавлением **j** в качестве суффикса (перемножаются мнимые единицы):

```
c=3-2j
print(c-4j+8) # в команде print можно выполнять действия
```

Тип **bool**. Подтип целочисленного типа для обозначения логических величин, которое принимает два значения: **True** (истина) и **False** (ложь).

```
for k in (True, False):
   for m in (False, True):
      print (k, m, ":", i and j, i or j, not i)
```

Результат предыдущего кода будет:
True False : True True False
True True : True True False
False False : True True False
False True : True True False

В Python строки бывают двух типов: **string** и **Unicode.** Строки-константы можно задать в программе с помощью строковых литералов апострофа **(')** или двойных кавычек **(")**. Управляющие последовательности внутри строковых литералов задаются обратной косой чертой **(\ )**. Пример:

```
str1 = "первая строка"
str2 = 'вторая строка\nи перевод на новую строку'
str3 = '\u043f\u0440\u0438\u0432\u0435\u0442'   # используя unicode
print(str1+'\n',str2+'\n',str3+'\n')
```

Набор операций над строками включает конкатенацию **"+"**, повтор **"*"**, форматирование **"%"**. Полный набор методов можно получить в документации.

```
print ("Str1" + "Str2")
print ("Str"*7)
print ("%s %f" % ("Добавим константу=", 273.15))
```

Для представления последовательности разнородных объектов используется тип кортеж **tuple**. Литерал кортежа обычно записывается в круглых скобках, но можно, если не возникают неоднозначности, писать и без них:

```
for i in "один", "два", "три": # цикл по значениям кортежа
   print (i)
one_item = (1,) # один элемент
empty = ()      # пустой кортеж
t1 = (0.3, 3.4, 1.8, 7.3)
t2 = 0.3, 3.4, 1.8, 7.3 # без скобок
print (one_item,empty,t1,t2)

tp = (3, [5, 2, 3.1], 1)
print(type(tp))   # <type 'tuple'>
```

В Python вместо массивов используются списки **list**, записываемых в квадратных скобках или посредством списковых включений.

```
lst1 = [30, 1.2, 8.3,]
lst2 = [k**2 for k in range(20) if k % 4 == 1]
lst3 = list("список из букв")
print ('\n',lst1,'\n',lst2,'\n',lst3)
```

Для того, чтобы получить отдельный элемент списка используются индексы, в квадратных скобках. Индексы последовательностей в Python начинаются **с нуля**. Отрицательные индексы служат для отсчета элементов с конца последовательности (-1 это последний элемент списка):

```
lst = [7.5, 2, 7, 3.6, 4.9]
print (lst[1], lst[0], lst[-1])
```

Словарь **dict** - это изменчивая структура данных для хранения пар ключ-значение, где значение однозначно определяется ключом. В качестве ключа может выступать неизменчивый тип данных (число, строка, кортеж и т.п.). Порядок пар ключ-значение произволен.

```
dct1 = {1: 'один', 2: 'два', 3: 'три', 4: 'четыре'}
dct2 = {0: 'zero'}
print (dct1[2]) # определяем значение словаря по ключу
dct1[0] = 0 # присваиваеваем значение по ключу
print (dct1)
del dct1[0] # удаляется пара ключ-значение
print (dct1)
# цикл по всему словарю
for key, val in dct1.items():
    print (key, val)
# цикл по ключам словаря
for key in dct1.keys():
    print (key, dct1[key])
# цикл по значениям словаря
for val in dct1.values():
    print (val)
dct1.update(dct2) # пополняется словарь из другого
print (dct1)
print (len(dct1)) # количество пар в словаре
```

Словарь (dict) изменяем — можно добавлять/удалять новые пары ключ: значение; значения элементов словаря — изменяемые и не уникальные; а вот ключи — не изменяемые и уникальные, поэтому, например, мы не можем сделать ключом словаря список, но можем кортеж. Из уникальности ключей, так же следует уникальность элементов словаря — пар ключ: значение. Скобки {} без значений **создают словарь**, а со значениями, в зависимости от синтаксиса могут создавать как множество, так и словарь:

```
d = {}
print(type(d))    # <class 'dict'>
s = {1, 2, 3}
print(type(s))    # <class 'set'>
n = {'d': 1, 's': 2}
print(type(n))    # <class 'dict'>
```

Тип **file.** Объекты этого типа предназначены для работы с внешними данными. Файловые объекты поддерживают основные методы: read(), write(), readline(), readlines(), seek(), tell(), close() и т.п. Рассмотрим код для копирования информации из одного файла в другой. Для этого необходимо в рабочей директории поместить два файла *file1.txt и file2.txt.*

```
f1 = open("file1.txt", "r")
f2 = open("file2.txt", "w")
for line in f1.readlines():
   f2.write(line)
f2.close()
f1.close()
```

На рис. 1 представлены все виды коллекций, представленных в Python.

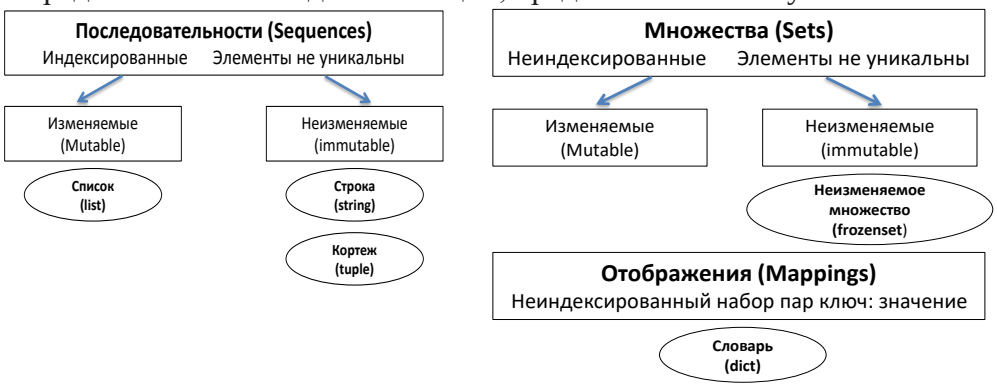


Рис. 1 Коллекции в Python.

Под **индексированностью** на рис. 1 понимается то, что у каждого элемента коллекции имеется свой порядковый номер — индекс. Это позволяет обращаться к элементу по его порядковому индексу, проводить слайсинг («нарезку»), когда необходимо выбрать часть коллекции. **Уникальность** – каждый элемент коллекции может встречаться в ней только один раз. Это порождает требование неизменности используемых типов данных для каждого элемента. **Изменяемость коллекции** — позволяет добавлять в коллекцию новых членов или удалять их после создания коллекции. На рис. 2 представлена таблица в которой приведено обобщение свойств встроенных коллекций в Python.

Тип коллекции	Изменяемость	Интегрированность	Уникальность	Как создаем
Список (list)	+	+	-	*[]* *list()*
Кортеж (tuple)	-	+	-	*(),* *tuple()*
Строка (string)	-	+	-	*‘’* *“”*
Множество (set)	+	-	+	**{elm1,elm2}** **Set()**
Неизменяемое множество (frozenset)	-	-	+	**frosenset()**
Словарь (dict)	**+ элементы** **- ключи** **+ значения**	-	**+ элементы** **+ ключи** **- значения**	**{}** **{key: value1,}** **dict()**

Рис. 2. Обобщение свойств, встроенных коллекций в сводной таблице

1.1.2. Общие подходы к работе с коллекциями.

Рассмотрим, что можно делать с любой стандартной коллекцией независимо от её типа:

 # список и словарь:

```
my_list = ['a', 'b', 'c']
print(type(my_list))
print(my_list)
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(type(my_dict))
print(my_dict)
```

Функции min(), max(), sum()

Поиск минимального и максимального значения осуществляется с помощью функции min() и max() соответственно. Если все элементы числовые, то можно произвести суммирование элементов с помощью функции sum().

```
print(min(my_list))
print(sum(my_dict.values()))
```

Подсчёт количества членов коллекции с помощью функции len()

```
print(len(my_list)) # 3
print(len(my_dict)) # 3 - для словаря пара ключ-значение считаются одним элементом.
print(len('ab c')) # 4 - для строки элементом является 1 символ
```

Проверка принадлежности элемента данной коллекции с помощью оператора in

x in s –вернет True, если элемент входит в коллекцию s и False – если не входит.

x not in s – также является вариантом проверки на принадлежность в коллекции.

```
my_list = ['a', 'b', 'c']
print('a' in my_list)          # True
print('a' not in my_list)      # False
```

Для словаря возможны следующие варианты:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
print('a' in my_dict)             # True - без указания метода поиск по ключам
print('a' in my_dict.keys())      # True - аналогично примеру выше
print('a' in my_dict.values())    # False - так как 'a' — ключ, не значение
print(1 in my_dict.values())      # True
```

Можно проверять пары.

```
print('abc' in 'abcd')    # True
print(('a',1) in my_dict.items())   # True
print(('a',2) in my_dict.items())   # False
```

Обход всех элементов коллекции в цикле for in.

В данном случае, в цикле будут последовательно перебираться элементы коллекции, пока не будут перебраны все из них.

```
for elm in my_list:
    print(elm)
```

У прохода в цикле по словарю есть свои особенности:

```
for elm in my_dict:
        print(elm) # равносильно for elm in my_dict.keys()
for elm in my_dict.values():
        print(elm) # по значениям
```

Но чаще всего нужны пары ключ (key) — значение (value). Проход по .items() возвращает кортеж (ключ, значение), который присваивается кортежу переменных key, value.

```
for key, value in my_dict.items():
   print(key, value)
```

Важно не менять количество элементов коллекции в теле цикла во время итерации по этой же коллекции, это вызывает не всегда очевидные ошибки.

Рассмотрим другие методы у коллекций:

.count() — метод подсчета определенных элементов для неуникальных коллекций (строка, список, кортеж), возвращает сколько раз элемент встречается в коллекции.

```
my_list = [0, 1, 2, 2, 2, 3]
print(my_list.count(2))    # 3 экземпляра элемента равного 2
print(my_list.count(5))    # 0 - нет элемента в коллекции
```

.index() — возвращает минимальный индекс переданного элемента для коллекций (строка, список, кортеж), напомним, что индексы начинаются с нуля

```
my_list = [0, 1, 2, 2, 3, 2]
print(my_list.index(2))  # первый элемент 2 находится по индексу 1
```

.copy() — метод возвращает неглубокую (не рекурсивную) копию коллекции (список, словарь, оба типа множества).

```
set1 = {0, 1, 2, 3}
set2 = set1.copy()
print(set1 == set1)  # True - коллекции равны
print(set2 is set1)  # False - коллекции не идентичны - разные объекты с разными id
```

.clear() — метод изменяемых коллекций (список, словарь, множество), удаляющий из коллекции все элементы и превращающий её в пустую коллекцию.

```
set1 = {0, 1, 2, 3}
print(set1)
set1.clear()
print(set1)  # set()
```

Тип коллекции	.count()	.index()	.copy()	.clear()
Список (list)	+	+	+ (python>=3.3)	+ (python>=3.3)
Кортеж (tuple)	+	+	-	-
Строка	+	+	-	-

(string)				
Множество (set)	-	-	+	+
Неизменяемое множество (frozenset)	-	-	+	-
Словарь (dict)	-	-	+	+

Рис. 3. Применимости методов, в зависимости от типа коллекции.

Особые методы сравнения множеств (set, frozenset):

set_a.isdisjoint(set_b) — истина, если set_a и set_b не имеют общих элементов.

set_b.issubset(set_a) — если все элементы множества set_b принадлежат множеству set_a, то множество set_b целиком входит в множество set_a и является его подмножеством (set_b — подмножество)

set_a.issuperset(set_b) — соответственно, если условие выше справедливо, то set_a — надмножество

```
set_a = {1, 2, 3}
set_b = {2, 1}  # порядок элементов не важен!
set_c = {4}
set_d = {1, 2, 3}
print(set_a.isdisjoint(set_c))  # True - нет общих элементов
print(set_b.issubset(set_a)) # True-set_b целиком входит в set_a, значит set_b - подмножество
print(set_a.issuperset(set_b)) # True - set_b целиком входит в set_a, значит set_a - надмножество
```

При равенстве множеств они одновременно и подмножество и надмножество друг для друга:

```
print(set_a.issuperset(set_d))  # True
print(set_a.issubset(set_d))    # True
```

1.1.3. Конвертация одного типа коллекции в другой

Один тип коллекции можно конвертировать в другой тип коллекции. При преобразовании в множество теряются дублирующие элементы, так как множество содержит только уникальные элементы. При конвертации индексированной коллекции в неиндексированную теряется информация о порядке элементов, а в некоторых случаев она может быть критически важной!

```
my_tuple = ('a', 'b', 'c','a','b')
my_list = list(my_tuple)
my_set = set(my_tuple)                  # теряем индексы и дубликаты элементов!
my_frozenset = frozenset(my_tuple)      # теряем индексы и дубликаты элементов!
print(my_list, my_set, my_frozenset)    # ['a', 'b', 'a'] {'a', 'b'} frozenset({'a', 'b'})
```

После конвертации в неизменяемый тип, мы больше не сможем менять элементы коллекции — удалять, изменять, добавлять новые. Это может привести к ошибкам в наших функциях обработки данных, если они были написаны для работы с изменяемыми

коллекциями. Способом выше не получится создать словарь, так как он состоит из пар ключ: значение. Это ограничение можно обойти, создав словарь, комбинируя ключи со значениями с помощью функции **zip()**:

```
my_keys = ('a', 'b', 'c')
my_values = [1, 2]      # Если количество элементов разное -
                # будет отработано пока хватает на пары - лишние отброшены
my_dict = dict(zip(my_keys, my_values))
print(my_dict)        # {'a': 1, 'b': 2}
```

Создаем строку из другой коллекции:

```
my_tuple = ('a', 'b', 'c')
my_str = ''.join(my_tuple)
print(my_str)      # abc
```

Если Ваша коллекция содержит изменяемые элементы (например, список списков), то ее нельзя конвертировать в не изменяемую коллекцию, так как ее элементы могут быть только не изменяемыми!

```
my_list = [1, [2, 3], 4]
my_set = set(my_list)   # TypeError: unhashable type: 'list'
```

1.1.4. Индексированные коллекции. Синтаксис и особенность среза.

Рассмотрим **индексированные коллекции** (их еще называют **последовательности** — **sequences**) — **список (list), кортеж (tuple), строку (string).** Под индексированностью имеется ввиду, что элементы коллекции располагаются в определённом порядке, каждый элемент имеет свой индекс от 0 до индекса на единицу меньшего длины коллекции (т.е. len(mycollection)-1). Для всех индексированных коллекций можно получить значение элемента по его индексу в квадратных скобках. Можно задавать отрицательный индекс, это значит, что будем находить элемент с конца считая обратном порядке. При задании отрицательного индекса, последний элемент имеет индекс -1, предпоследний -2 и так далее до первого элемента индекс которого равен значению длины коллекции с отрицательным знаком, то есть (-len(mycollection).

```
my_str = "abcd"
print(my_str[0])                # первый элемент
print(my_str[-1])               # последний элемент
print(my_str[len(my_str)-1])    # последний элемент
print(my_str[-2])               # d - предпоследний элемент
```

Коллекции могут иметь несколько уровней вложенности, как список списков в примере ниже. **Для перехода на уровень глубже ставится вторая пара квадратных скобок и так далее.**

```
lst = [[1, 2, 3], ['a', 'b', 'c']]
print(lst[0])      # первый вложенный список
print(lst[0][0])   # первый элемент первого вложенного списка
print(lst[1][-1])  # последний элемент второго вложенного списка
```

Поскольку кортежи и строки неизменяемые коллекции, то по индексу мы можем брать элементы, но не менять:

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[0])        # 1
my_tuple[0] = 100         # TypeError: 'tuple' object does not support item assignment
```

Для списка, если взятие элемента по индексу располагается в левой части выражения, а далее идёт оператор присваивания, то мы задаём новое значение элементу с этим индексом. Для такого присвоения, элемент уже должен существовать в списке, нельзя таким образом добавить элемент на несуществующий индекс.

```
my_list = [1, 2, 3, [4, 5]]
my_list[0] = 10
my_list[-1][0] = 40
print(my_list)              # [10, 2, 3, [40, 5]]

my_list = [1, 2, 3, 4, 5]
my_list[5] = 6      # IndexError: list assignment index out of range
```

На практике часто необходимо получить некоторый набор значений из списка, ограниченный определенными простыми правилами — например первые три или последние два, или каждый второй элемент — в таких задачах, вместо перебора в цикле намного удобнее использовать так называемый **срез (slice, slicing)**. Следует помнить, что, взяв элемент по индексу или срезом мы не как не меняем исходную коллекцию, мы просто скопировали ее часть для дальнейшего использования. Синтаксис среза представлен ниже:

```
my_collection[start:stop:step]  # старт, стоп и шаг
```

Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.

Отрицательное значение шага — перебор ведём в обратном порядке справа налево.

Если не указан старт [:stop:step] – начинаем с самого края коллекции, то есть с первого элемента (включая его), если шаг положительный или с последнего (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).

Если не указан стоп [start::step] — идем до самого края коллекции, то есть до последнего элемента (включая его), если шаг положительный или до первого элемента (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).

Если шаг равен одному, т.е. step = 1, то перебор слева направо указывать не обязательно — это значение шага по умолчанию. В таком случае достаточно указать [start:stop]. Чтобы взять коллекцию целиком достаточно указать [:]. При срезе, первый индекс входит в выборку, а второй нет. Можно представить данное важное замечание в виде формулы **[start, stop)**:

```
[ <первый включаемый> : <первый НЕ включаемый> : <шаг> ]
```

Поэтому, например, mylist[::-1] не идентично mylist[:0:-1], так как в первом случае мы включим все элементы, а во втором дойдем до 0 индекса, но не включим его!

Примеры расчета срезов приведены на рис. 4.

Последовательность	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**Результат**

Индексы	0(-7)	1(-6)	2(-5)	3(-4)	4(-3)	5(-2)	6(-1)	
[:] (→)	+	+	+	+	+	+	+	**abcdefg**
[::-1] (←)	+	+	+	+	+	+	+	**gfedcba**
[::2] (→)	+		+		+		+	**aceg**
[1::2] (→)		+		+		+		**bdf**
[:1]	+							**a**
[-1:]							+	**g**
[3:4]				+				**d**
[-3:] (→)					+	+	+	**efg**
[-3:1:-1] (←)			+	+	+			**edc**
[2:5] (→)			+	+	+			**cde**

Рис. 4 Примеры расчета срезов.

```
col = 'abcdefg'
print(col[:])      # abcdefg
print(col[::-1])   # gfedcba
print(col[::2])    # aceg
print(col[1::2])   # bdf
print(col[:1])     # a
print(col[-1:])    # g
print(col[3:4])    # d
print(col[-3:])    # efg
print(col[-3:1:-1]) # edc
print(col[2:5])    # cde
```

Можно задать константы с именованными срезами с пользованием специальной функции **slice()**. Переменная **None** соответствует опущенному значению по умолчанию. То есть [:2] становится **slice(None, 2)**, а [1::2] становится slice(1, None, 2).

```
person = ('Алиса', 'Алина', "March", 18, 2000)
Mame, Data = slice(None, 2), slice(2, None)
print(person[Mame])      # ('Alex', 'Smith')
print(person[Data])  # ('May', 10, 1980)

my_list = [1, 2, 3, 4, 5, 6, 7, 8]
slc = slice(1, None, 2)
print(my_list[slc])    # [2, 4, 6, 8]
```

Важный момент, с помощью среза можно не только получать копию коллекции, но в случае списка можно также менять значения элементов, удалять и добавлять новые. Даже если хотим добавить один элемент, необходимо передавать итерируемый объект, иначе будет ошибка: TypeError: can only assign an iterable

```
my_list = [1, 2, 3, 4, 5]
# my_list[1:2] = 20    # TypeError: can only assign an iterable
my_list[1:3] = [35]    # Вот теперь все работает
print(my_list)         #

my_list = [1, 2, 3, 4, 5]
```

```
my_list[1:3] = [35, 45]
print(my_list)        #
my_list[1:3] = [0]      # заменим два элемента на один
print(my_list)        # [1, 0, 4, 5]
my_list[2:] = [20, 30, 40]   # или два элемента на три
print(my_list)             # [1, 0, 20, 30, 40]
```

Можно просто удалить часть последовательности

```
my_list = [1, 2, 3, 4, 5]
my_list[:2] = []    # или del my_list[:2]
print(my_list)     # [3, 4, 5]
```

Для вставки одиночных элементов лучше использовать методы списка **.append() и .insert()**. Обращение по индексу по сути является частным случаем среза, когда мы обращаемся только к одному элементу, а не диапазону. **Обращение к несуществующему индексу** коллекции вызывает ошибку:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[-10])      # IndexError: list index out of range
print(my_list[10])       # IndexError: list index out of range
```

В случае **выхода границ среза за границы коллекции** никакой ошибки не происходит:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0:10])      # [1, 2, 3, 4, 5] — отработали в пределах коллекции
print(my_list[10:100])   # [] - таких элементов нет — вернули пустую коллекцию
```

Для сортировки элементов можно использовать функцию **sorted()**. Данная функция:
-не меняет исходную коллекцию, а возвращает новый список из ее элементов;
-не зависимо от типа исходной коллекции, вернётся список (list) ее элементов;
-не меняет исходную коллекцию, ее можно применять к неизменяемым коллекциям;
-можно применять к неиндексированным коллекциям;
-имеет дополнительные не обязательные аргументы:
reverse=True — сортировка в обратном порядке
key=funcname (начиная с Python 2.4) — сортировка с помощью специальной функции funcname, она может быть как стандартной функцией Python, так и специально написанной вами для данной задачи функцией и лямбдой.

```
my_list = [1, 7, 3.5, 2.1, 4.8]
my_list_sorted = sorted(my_list)
print(my_list_sorted)
```

```
my_set = {1, 7, 3.5, 2.1, 4.8}
my_set_sorted = sorted(my_set, reverse=True)
print(my_set_sorted)
```

Пример сортировки списка строк по длине len() каждого элемента:

```
my_files = ['cats.jpg', 'dog.png', 'apple.bmp', 'tv.gif']
```

```
my_files_sorted = sorted(my_files, key=len)
print(my_files_sorted)
```

Функция **reversed()** применяется для последовательностей и работает по другому:

-возвращает генератор списка, а не сам список; если нужно получить не генератор, а готовый список, результат можно обернуть в list() или же вместо **reversed()** воспользоваться срезом [: :-1];

-не сортирует элементы, а возвращает их в обратном порядке, то есть читает с конца списка;

-если коллекция неиндексированная — мы не можем вывести её элементы в обратном порядке и эта функция к таким коллекциям не применима — получим «TypeError: argument to reversed() must be a sequence»;

-не позволяет использовать дополнительные аргументы — будет ошибка «TypeError: reversed() does not take keyword arguments».

```
my_list = [2, 5, 1, 7, 3]
my_list_sorted = reversed(my_list)
print(my_list_sorted)        # <listreverseiterator object at 0x7f8982121450>
print(list(my_list_sorted))   # [3, 7, 1, 5, 2]
print(my_list[::-1])         # [3, 7, 1, 5, 2] - тот же результат с помощью среза
```

У списка есть особые методы **.sort()** и **.reverse()** которые делают тоже самое, что и функции **sorted()** и **reversed()**, но меняют сам исходный список, а не генерируют новый; Возвращают **None**, а не новый список; поддерживают те же дополнительные аргументы.

```
my_list = [2, 5, 1, 7, 3]
my_list.sort()
print(my_list)        # [1, 2, 3, 5, 7]
```

Частая ошибка, которая не является ошибкой для интерпретатора, но приводит не к тому результату, который хотят получить.

```
my_list = [2, 5, 1, 7, 3]
my_list = my_list.sort()
print(my_list)        # None
```

В сортировке словаря есть свои особенности, вызванные тем, что элемент словаря — это пара ключ: значение. Сохранить данные сортированными в самом стандартном словаре не получится, они в нем, как и других неиндексированных коллекциях находятся в произвольном порядке. Для словаря можно использовать следующие функции:

sorted(my_dict) — когда мы передаем в функцию сортировки словарь без вызова его дополнительных методов — идёт перебор только ключей, сортированный список ключей нам и возвращается;

sorted(my_dict.keys()) — тот же результат, что в предыдущем примере, но прописанный более явно;

sorted(my_dict.items()) — возвращается сортированный список кортежей (ключ, значение), сортированных по ключу;

sorted(my_dict.values()) — возвращается сортированный список значений

```
my_dict = {'a': 1, 'c': 3, 'e': 5, 'f': 7, 'b': 2}
mysorted = sorted(my_dict)
```

```
print(mysorted)
mysorted = sorted(my_dict.items())
print(mysorted)
mysorted = sorted(my_dict.values())
print(mysorted)
```

Отдельные сложности может вызвать сортировка словаря не по ключам, а по значениям, если нам не просто нужен список значений, и именно выводить пары в порядке сортировки по значению. Для решения этой задачи можно в качестве специальной функции сортировки передавать функцию **lambda x: x[1]** которая из получаемых на каждом этапе кортежей (ключ, значение) будет брать для сортировки второй элемент кортежа. Дополнительные детали и примеры использования параметра key представлены в [4]:

```
population = {"Shanghai": 24256800, "Karachi": 23500000, "Beijing": 21516000, "Delhi": 16787941} # отсортируем по возрастанию населения:
population_sorted = sorted(population.items(), key=lambda x: x[1])
print(population_sorted)
# [('Delhi', 16787941), ('Beijing', 21516000), ('Karachi', 23500000), ('Shanghai', 24256800)]
```

Допустим данные нужно отсортировать сначала по столбцу A по возрастанию, затем по столбцу B по убыванию, и наконец по столбцу C снова по возрастанию. Если данные в столбце B числовые, то при помощи подходящей функции в key можно поменять знак у элементов B, что приведёт к необходимому результату. Функция сортировки sort() в Python устойчивая (начиная с Python 2.2), то есть она не меняет порядок «одинаковых» элементов. Поэтому можно просто отсортировать три раза по разным ключам:

```
data.sort(key=lambda x: x['C'])
data.sort(key=lambda x: x['B'], reverse=True)
data.sort(key=lambda x: x['A'])
```

Дополнительная информация по устойчивости сортировки представлена в работе [5]. Рассмотрим способы объединения строк, кортежей, списков, словарей без изменения исходных коллекций — когда из нескольких коллекций создаётся новая коллекция того же типа без изменения изначальных. Объединение строк (string) и кортежей (tuple) возможна с использованием оператора сложения «+»

```
str1 = 'ab1'
str2 = '34df'
str3 = str1 + str2
print(str3)
tpl1 = (1, 2, 3, 4, 5)
tpl2 = (6, 7)
tpl3 = tpl1 + tpl2
print(tpl3)
```

Для объединения списков возможны три варианта без изменения исходного списка:

```
a = [1, 2, 3]
b = [4, 5]
c = a + b
print(a, b, c)      # [1, 2, 3]  [4, 5]  [1, 2, 3, 4, 5]
```

Добавляем второй список как один элемент без изменения исходного списка (аналог метода.**append()**, но без изменения исходного списка):

```
a = [1, 2, 3]
b = [4, 5]
c = a + [b]
print(a, b, c)     # [1, 2, 3]  [4, 5]  [1, 2, 3, [4, 5]]
```

В Питоне 3.5 появился новый более изящный способ:

```
a, b = [1, 2, 3], [4, 5]
c = [*a, *b]  # работает на версии питона 3.5 и выше
print(c)      # [1, 2, 3, 4, 5]
```

Объединить два словаря с помощью оператора “+” Python не получится, это вызовет ошибку «TypeError: unsupported operand type(s) for +: 'dict' and 'dict'». Для этого воспользуемся методами .copy() и .update():

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = dict1.copy()
dict3.update(dict2)
print(dict3)
```

В Питоне 3.5 появился новый более изящный способ:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {**dict1, **dict2}
print(dict3)                # {'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

Для обоих типов множеств (**set, frozenset**) возможны различные варианты комбинации множеств (исходные множества при этом не меняются — возвращается новое множество).

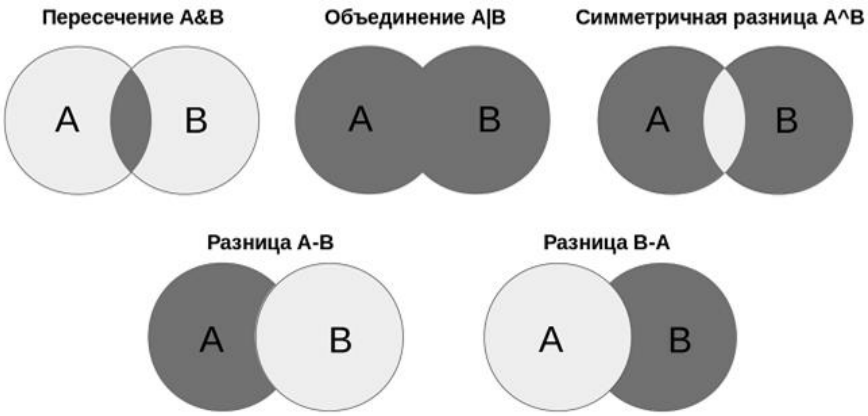


Рис. 5 Пересечение, объединение и разница множеств.

```
# Зададим исходно два множества
a = {'a', 'b'}
b = {'b', 'c'}
```

c = {'b', 'd'}

Пересечение (intersection)	**Объединение (union)**	**Разница (difference)**	**Симметричная разница (symmetric_difference)**	**Объединение списка, словаря и множества**
c = a.intersection(b) *#или c =b.intersection(a)* *# возможно так:* *c = a & b* *print(c) # {'b'}* *a = {'a', 'b'}* *b = { 'b', 'c'}* *c = { 'b', 'd'}* *# Первый вариант* *d = a.intersection(b, c)* *# Второй вариант* *d = set.intersection(a, b, c)* *print(d) # {'b'}*	*c = a.union(b)* *# или c =* *b.union(a)* *# c=a+b не работает* *# или c = a | b* *print(c)* *# {'a', 'c', 'b'}*	*c=a.difference(b)* *#или c=a-b* *print(c)* *# {'a'}* *c=b.difference(a)* *#или c = b - a* *print(c) # {'c'}*	*c =* *b.symmetric_difference(a)* *# или c =* *a.symmetric_difference(b)* *# возможно так:* *c = b ^ a* *print(c) # {'a', 'c'}*	*a.extend(b)* *# или a += b)* *print(a, b)* *# [1, 2, 3, 4, 5] [4, 5]* *a.append(b)* *# или a += [b]* *print(a, b)* *# [1,2,3,[4, 5]] [4, 5]* *dict1 = {'a': 1, 'b': 2}* *dict2 = {'a':100, 'c':3, 'd':4}* *dict1.update(dict2)* *print(dict1)* *# {'a': 100, 'c': 3, 'b': 2, 'd': 4}*

Для изменяемого множества (set) кроме операций, описанных в предыдущем разделе, также возможны их аналоги, но уже с изменением исходного множества — эти методы заканчиваются на _update. Рассмотрим следующие методы:

.difference_update()	**.intersection_update()**	**.symmetric_difference_update**
a = {'a', 'b'} *b = { 'b', 'c'}* *a.difference_update(b)* *print(a, b) # {'a'} {'b', 'c'}*	*a = {'a', 'b'}* *b = { 'b', 'c'}* *a.intersection_update(b)* *print(a, b) # {'b'} {'b', 'c'}*	*a = {'a', 'b'}* *b = { 'b', 'c'}* *a.symmetric_difference_update(b)* *print(a, b) # {'c', 'a'} {'c', 'b'}*
a = {'a', 'b'} *b = { 'b', 'c'}* *b.difference_update(a)* *print(a, b) # {'a', 'b'} {'c'}*	*a = {'a', 'b'}* *b = { 'b', 'c'}* *b.intersection_update(a)* *print(a, b) # {'b', 'a'} {'b'}*	*a = {'a', 'b'}* *b = { 'b', 'c'}* *b.symmetric_difference_update(a)* *print(a, b) # {'a', 'b'} {'c', 'a'}*

1.1.5. Добавление и удаление элементов изменяемых коллекций.

Добавление и удаление элементов в коллекцию возможно только для изменяемых коллекций: списка (list), множества (только set, не frozenset), словаря (dict). Причём для списка, который является индексированной коллекцией, также важно на какую позицию мы добавляем элемент. Методы добавления элементов представлены в таблице:

Метод/оператор	**Описание**	**Коллекция**		
		list	**dict**	**set**
Добавление элемента				
.add()	Добавляем элемент в множество. Если такой уже существует - ничего не происходит.	**-**	**-**	**+**
.append()	Добавляем элемент в конец списка.	**+**	**-**	**-**
.extend()	Расширяем список добавлением элементов из переданного итерируемого.	**+**	**-**	**-**
.insert()	Вставляем элемент в список перед переданным индексом.	**+**	**-**	**-**
[i:j:k]	Добавление элементов списка срезом в указанное место, возможно с заменой существующих.	**+**	**-**	**-**

Методы удаления элементов представлены в таблице ниже:

Метод/оператор	Описание	Коллекция		
		list	dict	set
Удаление элемента				
.clear()	Удаляем из коллекции все элемента и превращаем ее в пустою коллекцию.	+	+	+
.discard()	Аналог .remove(), но работает только на множестве и не выдает ошибок в случае отсутствия элемента.	-	-	+
.pop()	Удаляет и возвращает значения элемента по указанному индексу (для списка) или просто случайны элемент (для множества). В случае отсутствия указанного элементаа или если множество пустое появится ошибка: *ValueError* (для списка) или *KeyError* (для множества).	+	+	+
.popitem()	Аналог .pop(), но работает только на словаре и вовзращает не значение, а случайный кортеж (ключ, значение) или *KeyError*, если словарь пустой.	-	+	-
.remove()	Удаляет элемент по значению – для списка удаляется только первый с таким значением. Во множестве элемент в любом случае уникален. В случае отсутствия элемента с таким значением появится ошибка: *ValueError* (для списка) или *KeyError* (для множества).	+	-	+
[i:j:k]	Удаляем часть элеметов списка срезом с помощью оператора del или присвоением им пустого списка [].	+	-	-
del	Удаляем элемент по индексу (для списка) или срезу (для списка) или ключу (для словаря), ничего не возвращаем. В случае отсуствия указанного элемент – ошибка: *IndexError* (для списка) или *KeyError* (для словаря).	+	+	-

Методы обновления элементов коллекции представлены в таблице ниже:

Метод/оператор	Описание	Коллекция		
		list	dict	set
Обновление коллекции				
.update()	В словаре для существующих ключей обновляются значения, для новых ключей – добавляется пара ключ: значение. Для множества происходит замена множества его объединением с переданными в метод множествами.	-	+	+

Особенности работы с изменяемой и не изменяемой коллекцией. Строка – неизменяемая коллекция — если мы ее меняем — мы создаем новый объект.

```
str1 = 'abcd'
print(str1, id(str1))      #
str1 += 'efg'
print(str1, id(str1))      #  новый объект, с другим id
```

Пример кода с двумя исходно идентичными строками.

```
str1 = 'abcd'
str2 = str1
print(str1 is str2)    # True две ссылки на один и тот же объект
```

```
str1 += 'efg'          # Теперь переменная str1 ссылается на другой объект
print(str1 is str2)    # False - теперь это два разных объекта
print(str1, str2)      # разные значения
```

Можно сравните данный пример с примером ниже:

```
list1 = [1, 2, 3]
list2 = list1
print(list1 is list2)    # True две ссылки на один и тот же объект!
# А дальше убеждаемся, насколько это важно:
list1 += [4]
print(list1, list2)      # [1, 2, 3, 4] [1, 2, 3, 4]
# изменилось значение переменных
```

Если необходима независимая копия:

```
list1 = [1, 2, 3]
list2 = list(list1)      # 1-й способ копирования
list3 = list1[:]         # 2-й способ копирования
list4 = list1.copy()     # 3-й способ копирования
print(id(list1), id(list2), id(list3), id(list4))
# мы создали 4 разных объекта

list1 += [4]             # изменяем исходный список
print(list1, list2, list3, list4)      # [1, 2, 3, 4] [1, 2, 3] [1, 2, 3] [1, 2, 3]
# изменив исходный объект, его копии остались не тронутыми
```

1.1.6. Генераторы выражений. Вложенные циклы и генераторы.

Для компактного и удобного способа генерации коллекций элементов, а также преобразования одного типа коллекций в другой служат **генераторы выражений**. Преимуществом использования генераторов является то, что используется удобный и короткий синтаксис. **Генератор коллекции** — обобщенное название для генератора списка (**list comprehension**), генератора словаря (**dictionary comprehension**) и генератора множества (**set comprehension**). **Выражение-генератор (generator expression)** — выражение в круглых скобках которое создает на каждой итерации новый элемент по правилам. На рис. 6 представлен синтаксис генератора.

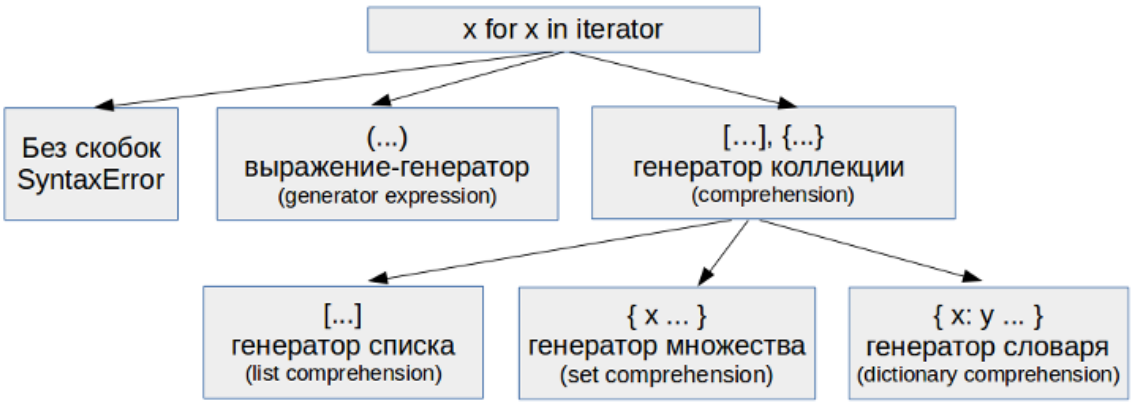


Рис. 6 Синтаксис генератора.

Общий синтаксиса выражения-генератора приведен на рис. 7. Можно отметить, что данный синтаксис одинаков и для выражения-генератора и для всех трех типов генераторов коллекций, разница заключается, в каких скобках он будет заключен.

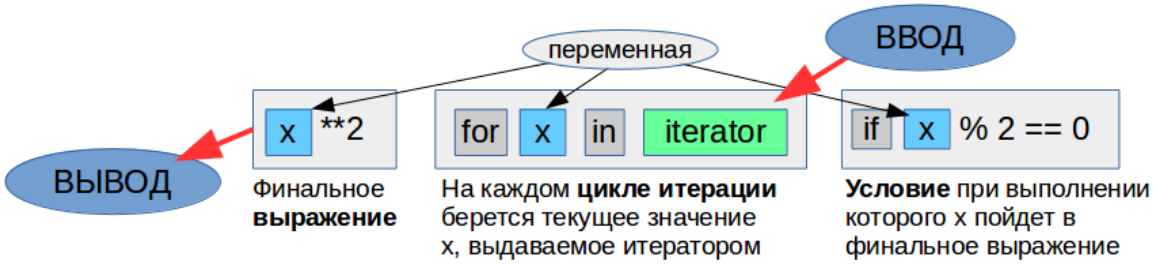


Рис. 7 Общие принципы построения генератора.

Общие принципы важные для понимания:

Ввод — это итератор —может быть функция-генератор, выражение-генератор, коллекция — любой объект поддерживающий итерацию по нему.

Условие — это фильтр при выполнении которого элемент пойдет в финальное выражение, если элемент ему не удовлетворяет, он будет пропущен.

Финальное выражение — преобразование каждого выбранного элемента перед его выводом или просто вывод без изменений.

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x for x in list_1]
print(list_2)                  # [-2, -1, 0, 1, 2, 3, 4, 5]
print(list_1 is list_2)          # это разные объекты
```

Мощь генераторов выражений заключается в том, что мы можем задавать условия для включения элемента в новую коллекцию и можем делать преобразование текущего элемента с помощью выражения или функции перед его выводом (включением в новую коллекцию). Добавим в предыдущий пример условие — брать только четные элементы.

```
# if x % 2 == 0 - остаток от деления на 2 равен нулю - число четное
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x for x in list_1 if x % 2 == 0]
print(list_2)   # [-2, 0, 2, 4]

list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x for x in list_1 if x % 2 == 0 and x > 0]
# берем те x, которые одновременно четные и больше нуля
print (list_2)   # [2, 4]
```

Можно посчитать квадраты значений каждого элемента:

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x**2 for x in list_1]
print(list_2)   # [4, 1, 0, 1, 4, 9, 16, 25]
```

Или посчитать длины строк с помощью функции len()

```
list_1 = ['a', 'abc', 'abcde']
list_2 = [len(x) for x in list_a]
```

```
print(list_3)   # [1, 3, 5]
```

Можно использовать ветвление выражения. Условия ветвления пишутся не после, а перед итератором. В данном случае if-else это не фильтр перед выполнением выражения, а ветвление самого выражения.

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x if x < 0 else x**2 for x in list_1]
print(list_2)   # [-2, -1, 0, 1, 4, 9, 16, 25]
```

Можно комбинировать фильтрацию и ветвление:

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [x**3 if x < 0 else x**2 for x in list_1 if x % 2 == 0]
print(list_2)   # [-8, 0, 4, 16]
```

решение предыдущей задачи без генератора:

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = []
for x in list_1:
    if x % 2 == 0:
        if x < 0:
            list_2.append(x ** 3)
        else:
            list_2.append(x ** 2)
print(list_2)   # [-8, 0, 4, 16]
```

Используя переносы строк внутри скобок можно сделать синтаксис генераторов выражений более легким для чтения:

```
numbers = range(10)
# Before
squared_evens = [n ** 2 for n in numbers if n % 2 == 0]

# After
squared_evens = [
    n ** 2
    for n in numbers
    if n % 2 == 0
]
```

Выражения-генераторы (generator expressions) доступны, начиная с Python 2.4. Основное их отличие от генераторов коллекций в том, что они выдают элемент по-одному, не загружая в память сразу всю коллекцию. Если мы создаем большую структуру данных без использования генератора, то она загружается в память целиком, соответственно, это увеличивает расход памяти, а в крайних случаях памяти может просто не хватить. В случае использования **выражения-генератора**, такого не происходит, так как элементы создаются по-одному, в момент обращения. Пример выражения-генератора:

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
```

```
my_gen = (i for i in list_1)    # выражение-генератор
print(next(my_gen))    # -2 - получаем очередной элемент генератора
print(next(my_gen))    # -1 - получаем очередной элемент генератора
```

Свойства выражения-генератора:

# my_gen = i for i in list_1 *#SyntaxError: invalid syntax*	Генератор нельзя писать без скобок — это синтаксическая ошибка
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_sum = sum(i for i in list_1)* *# my_sum = sum((i for i in list_1)) # можно так* *print(my_sum) # 12*	При передаче в функцию дополнительные скобки необязательны
# my_len = len(i for i in list_1) *# TypeError: object of type 'generator' has no len()*	Нельзя получить длину выражения-генератора.
print(my_gen) # <generator object <genexpr> at 0x7f162db32af0>	Нельзя распечатать элементы функцией.
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_gen = (i for i in list_1)* *print(sum(my_gen)) # 12* *print(sum(my_gen)) # 0*	После прохождения по выражению-генератору оно остается пустым.
import itertools *inf_gen = (x for x in itertools.count())* *# бесконечный генератор от 0 to бесконечности!*	Выражение-генератор может быть бесконечным. Можно получить «эффект» как от бесконечного цикла.
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_gen = (i for i in list_1)* *my_gen_sliced = my_gen[1:3]* *# TypeError: 'generator' object is not subscriptable*	К выражению-генератору не применимы срезы.
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_gen = (i for i in list_1) # выражение-генератор* *my_list = list(my_gen)* *print(my_list) # [-2, -1, 0, 1, 2, 3, 4, 5]* *# список* *list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]* *my_list = list(i for i in list_1)* *print(my_list) # [-2, -1, 0, 1, 2, 3, 4, 5]* *# кортеж* *my_tuple = tuple(i for i in list_1)* *print(my_tuple) # (-2, -1, 0, 1, 2, 3, 4, 5)* *# множество* *my_set = set(i for i in list_1)* *print(my_set) # {0, 1, 2, 3, 4, 5, -1, -2}* *# неизменное множество* *my_frozenset = frozenset(i for i in list_1)* *print(my_frozenset) # frozenset({0, 1, 2, 3, 4, 5, -1, -2})*	Создание коллекций из выражения-генератора с помощью функций list(), tuple(), set(), frozenset(). Написание выражения-генератора сразу внутри скобок вызываемой функции создания коллекции.

В отличии от выражения-генератора, которое выдает значение по-одному, не загружая всю коллекцию в память, при использовании **генераторов коллекций**, коллекция генерируется сразу целиком. Соответственно, вместо особенности выражений-генераторов, перечисленных выше, такая коллекция будет обладать всеми стандартными свойствами характерными для коллекции данного типа.

Использование **генераторов коллекций**

list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_list = [i for i in list_1]* *print(my_list) # [-2, -1, 0, 1, 2, 3, 4, 5]* *my_list = [(i for i in list_1)]* *print(my_list)* *# [<generator object <genexpr> at 0x7fb81103bf68>]*	Генератор списка (list comprehension). Не пишите круглые скобки в квадратных!
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *my_set= {i for i in list_1}* *print(my_set)* *# {0, 1, 2, 3, 4, 5, -1, -2} - порядок случаен*	Генератор множества (set comprehension)
dict_abc = {'a': 1, 'b': 2, 'c': 3, 'd': 3} *dict_123 = {v: k for k, v in dict_abc.items()}* *print(dict_123) # {1: 'a', 2: 'b', 3: 'd'}* *# Обратите внимание, мы потеряли "c"! Так как значения были одинаковы, то, когда они стали ключами, только последнее значение сохранилось.*	Генератор словаря (dictionary comprehension).
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *dict_1 = {x: x**2 for x in list_1}* *print(dict_1) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, -2: 4, -1: 1, 5: 25}*	Словарь из списка.
*# dict_gen = (x: x**2 for x in list_1)* *# SyntaxError: invalid syntax* *dict_gen = ((x, x ** 2) for x in list_1)* *# Корректный вариант генератора-выражения для словаря* *# dict_1 = dict(x: x**2 for x in list_1)* *# SyntaxError: invalid syntax* *dict_1 = dict((x, x ** 2) for x in list_1)* *# Корректный вариант синтаксиса от @longclaps*	Синтаксис создания словаря.
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5] *# используем генератор прямо в .join() одновременно приводя элементы к строковому типу* *my_str = ''.join(str(x) for x in list_1)* *print(my_str) # -2-1012345*	Элементы коллекции для объединения в строку должны быть строками.

Иногда в условиях задачи нужна не проверка значения текущего элемента, а проверка на определенную периодичность, например, нужно брать каждый третий элемент. Для подобных задач можно использовать функцию **enumerate()**, задающую счетчик при обходе итератора в цикле. Функция **enumerate()** не обращается к каким-то внутренним атрибутам коллекции, а просто реализует счетчик обработанных элементов, поэтому ничего не мешает ее использовать для неупорядоченных коллекций не имеющих индексации.

Возможные варианты вызова функции enumerate():

- enumerate(iterator) без второго параметра считает с 0.
- enumerate(iterator, start) — начинает считать с значения start.

- enumerate() возвращает кортеж из порядкового номера и значения текущего элемента итератора. Кортеж в выражении-генераторе результате можно получить двумя способами:
- (i, j) for i, j in enumerate(iterator) — скобки в первой паре нужны
- pair for pair in enumerate(mylist) —работаем сразу с парой

Проиллюстрируем работу с индексами:

```
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_2 = [(i, x) for i, x in enumerate(list_1)]
print(list_2)  # [(0, -2), (1, -1), (2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5)]
list_1 = [-2, -1, 0, 1, 2, 3, 4, 5]
list_3 = [x for i, x in enumerate(list_1, 1) if i % 3 == 0]
print(list_3)  # [0, 3]
first_ten_even = [(i, x) for i, x in enumerate(range(10)) if x % 2 == 0]
print(first_ten_even)  # [(0, 0), (2, 2), (4, 4), (6, 6), (8, 8)]
```

Иногда бывает задача из очень большой коллекции или даже бесконечного генератора получить выборку первых нескольких элементов, удовлетворяющих условию. Если мы используем обычное генераторное выражение с условием ограничением по enumerate() индексу или срез полученной результирующей коллекции, то нам в любом случае придется пройти всю огромную коллекцию и потратить на это уйму компьютерных ресурсов. Выходом может быть использование **функции islice()** из пакета **itertools**.

```
import itertools
first_ten = (itertools.islice((x for x in range(1000000000) if x % 2 == 0), 10))
print(list(first_ten))  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Рассмотрим более комплексные варианты, когда у нас циклы или сами выражения-генераторы являются вложенными. Тут возможны несколько вариантов, со своими особенностями и сферой применения, чтобы не возникало путаницы, рассмотрим их по-отдельности, а после приведем общую схему.

Вложенные циклы

[expression for x in iter1 for y in iter2]

Для примера создадим словарь, используя кортежи координат как ключи, заполнив для начала его значения нулями.

```
rows = 1, 2, 3
cols = 'a', 'b'
my_dict = {(col, row): 0 for row in rows for col in cols}
print(my_dict)  # {('a', 1): 0, ('b', 2): 0, ('b', 3): 0, ('b', 1): 0, ('a', 3): 0, ('a', 2): 0}
```

Вложенные циклы for где внутренний цикл идет по результату внешнего цикла

[expression for x in iterator for y in x]

Рассмотрим пример, когда необходимо обходить двумерную структуру данных, превращая ее в одномерную. В данном случае, мы во внешнем цикле проходим по строкам, а во внутреннем по элементам каждой строки нашей двумерной структуры.

```
matrix = [[1, 2, 3, 4],
          [5, 6, 7, 8],
```

```
    [9, 10, 11, 12]]
flattened = [n for row in matrix for n in row]
print(flattened)
```

Вложенные генераторы

[[expression for y in iter2] for x in iter1]

Вложенными могут быть и сами генераторы. Такой подход применяется, когда нам надо строить двумерную структуру. В отличии от примеров выше с вложенными циклами, для вложенных генераторов, вначале обрабатывается внешний генератор, потом внутренний, то есть порядок идет справа-налево. Рассмотрим пример создания матрицы, состоящей из 8 столбцов и 4 строк.

```
w, h = 8, 4  # зададим ширину и высотку матрицы
matrix = [[0 for i in range(w)] for o in range(h)]
print(matrix)
```

После создания можем работать с матрицей как с обычным двумерным массивом.

Обходим двумерную структуру данных, сохраняя результат в другую двумерную структуру.

```
matrix = [[1, 2, 3, 4, 5], [5, 6, 7, 8, 9 ],[10, 20, 30, 40]]
squared = [[cell**2 for cell in row] for row in matrix]
# Возведем каждый элемент матрицы в квадрат:
print(squared)
```

Обобщим все вышеперечисленные варианты в одной схеме.

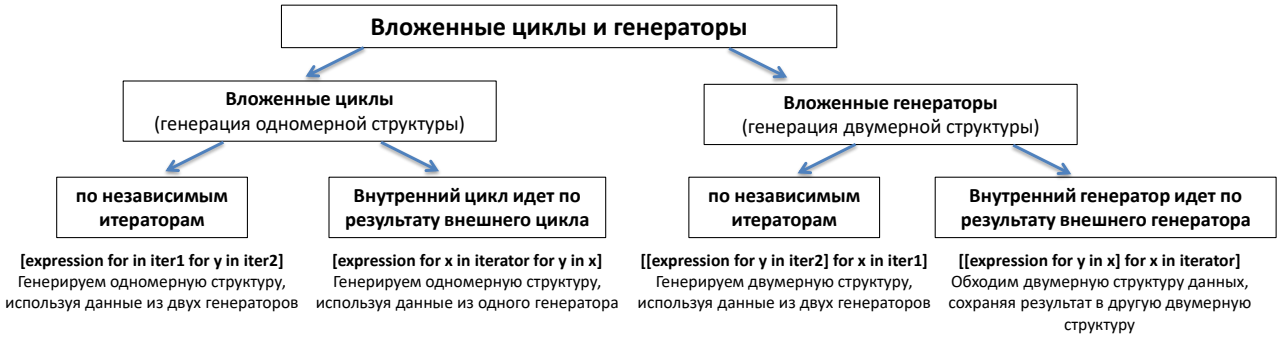


Рис. 8 Общие принципы построения вложенных циклов и генераторов.

Допустим у нас есть два таких генератора списков:

```
list_1 = [x for x in range(-2, 4)]
list_2 = [x**2 for x in list_1]
```

Данное выражение можно записать в одно выражение, подставив вместо list_1 его генератор списка:

```
list_3= [x**2 for x in [x for x in range(-2, 4)]]
print(list_3)  # [4, 1, 0, 1, 4, 9]
```

Преимущество комбинирования генераторов на примере сложной функции f(x)=u(v(x)):

```
list_3 = [t + t ** 2  for t in (x ** 3 + x ** 4 for x in range(-2, 4))]
```

Рассмотрим функцию **range()**, которая предназначена для создания арифметических последовательностей. В Python 3 range() возвращает генератор, который при каждом к нему обращении выдает очередной элемент. Основные параметры функции range().

- **range(stop)** — в данном случае с 0 до stop-1;

- **range(start, stop)** — Аналогично примеру выше, но можно задать начало отличное от нуля, можно и отрицательное;

- **range(start, stop, step)** — Добавляем параметр шага, который может быть отрицательным, тогда перебор в обратном порядке.

Примеры использования функции range():

```
print(list(range(5)))          # [0, 1, 2, 3, 4]
print(list(range(-2, 5)))      # [-2, -1, 0, 1, 2, 3, 4]
print(list(range(5, -2, -2)))  # [5, 3, 1, -1]
```

1.2. Задания

1. Вычислить сумму векторов $a = (5.2\ 3.1\ 8.9)$ и $b = (1.7\ 3.5\ 4.9)$
2. Вывести второй элемент вектора-строки v = (0.2 8.3 7.8 3.1 6.4), заменить четвертый элемент вектора-строки на 4.7, записать в массив v первый, пятый и третий элементы.
3. В массиве вектора-строки w = (1.8 6.4 9.3 0.5 2.1 3.7 2.9) заменить нулями элементы с третьего по седьмой, создать новый массив w1, используя элементы массива w со второго по пятый и составить массив w2, содержащий элементы w, кроме второго (используя сцепление строк).
4. Найти минимальный и максимальный элементы вектора z=[2.2; 4.1; 8.9; 3.2; 2.1; 5.8; 2.1; 2.1; 9.7]; найти минимальный и максимальный элементы вектора z и индекс (порядковый номер) максимального элемента (вызвать функцию max).
5. Упорядочить вектор-строку r = (–0.2 6.3 –9.4 3.8 7.4 0.1): (а) по возрастанию, (б) по убыванию
6. Перемножить вектор-строки v1 = (2 8 –4 3) и v2 = (–5 7 –3 1); (а) возвести во вторую степень вектор v1; (б) разделить вектор v1 на v2 и v2 на v1.
7. Присвоить строковой переменной str алфавит. Написать программу, которая выведет на экран (а) строку с каждой второй буквой из переменной str; (б) строку с каждой седьмой буквой из переменной str. (в) прочитать файл с текстом, из которого программа прочитает каждый второй и каждый седьмой символ и запишет текст в новый файл.

ОСНОВНЫЕ СТАНДАРТНЫЕ МОДУЛИ PYTHON

Одним из важных преимуществ языка Python является наличие большой библиотеки модулей и пакетов, входящих в стандартную поставку. В соответствии с модульным подходом к программированию большая задача разбивается на несколько более мелких, каждую из которых (в идеале) решает отдельный модуль. В разных методологиях даются различные ограничения на размер модулей, однако при построении модульной структуры программы важнее составить такую композицию модулей, которая позволила бы свести к минимуму связи между ними. Набор классов и функций, имеющий множество связей между своими элементами, было бы логично расположить в одном модуле. Модуль должен иметь удобный интерфейс: набор функций, классов и констант, который он предлагает своим пользователям. В программе на Python модуль представлен объектом-модулем, атрибутами которого являются имена, определенные в модуле:

```
import datetime
d1 = datetime.date(2004, 11, 20)
print(d1)
```

В данном примере импортируется модуль datetime. В результате работы оператора import в текущем пространстве имен появляется объект с именем datetime.

Модули для использования в программах на языке Python по своему происхождению делятся на обычные (написанные на Python) и модули расширения, написанные на другом языке программирования (как правило, на C). С точки зрения пользователя они могут отличаться разве что быстродействием. Бывает, что в стандартной библиотеке есть два варианта модуля: на Python и на C. Таковы, например, модули pickle и cPickle.

Модуль оформляется в виде отдельного файла с исходным кодом. Стандартные модули находятся в каталоге, где их может найти соответствующий интерпретатор языка. Пути к каталогам, в которых Python ищет модули, можно увидеть в значении переменной sys.path.

Подключение модуля к программе на Python осуществляется с помощью оператора import. У него есть две формы: import и from-import:

С помощью первой формы с текущей областью видимости связывается только имя, ссылающееся на объект модуля, а при использовании второй - указанные имена (или все имена, если применена *) объектов модуля связываются с текущей областью видимости. При импорте можно изменить имя, с которым объект будет связан, с помощью as. В первом случае пространство имен модуля остается в отдельном имени и для доступа к конкретному имени из модуля нужно применять точку. Во втором случае имена используются так, как если бы они были определены в текущем модуле:

В среде Python без дополнительных операций импорта доступно более сотни встроенных объектов, в основном, функций и исключений. Для удобства функции условно разделены по категориям:

1. Функции преобразования типов и классы: coerce, str, repr, int, list, tuple, long, float, complex, dict, super, file, bool, object
2. Числовые и строковые функции: abs, divmod, ord, pow, len, chr, unichr, hex, oct, cmp, round, unicode
3. Функции обработки данных: apply, map, filter, reduce, zip, range, xrange, max, min, iter, enumerate, sum
4. Функции определения свойств: hash, id, callable, issubclass, isinstance, type
5. Функции для доступа к внутренним структурам: locals, globals, vars, intern, dir
6. Функции компиляции и исполнения: eval, execfile, reload, __import__, compile
7. Функции ввода-вывода: input, raw_input, open
8. Функции для работы с атрибутами: getattr, setattr, delattr, hasattr
9. Функции-"украшатели" методов классов: staticmethod, classmethod, property

10. Прочие функции: buffer, slice

Уточнить назначение функции, ее аргументов и результата можно в интерактивной сессии интерпретатора Python: >>> help(len)

Функции работают с числовыми или строковыми аргументами. В следующей таблице даны описания этих функций. Примеры некоторых функций:

abs(x) Модуль числа x.

divmod(x, y) Частное и остаток от деления.

pow(x, y[, m]) Возведение x в степень y по модулю m.

round(n[, z]) Округление чисел до заданного знака после (или до) точки.

ord(s) Функция возвращает код (или Unicode) заданного ей символа в односимвольной строке.

chr(n) Возвращает строку с символом с заданным кодом.

len(s) Возвращает число элементов последовательности или отображения.

oct(n), hex(n) Функции возвращают строку с восьмеричным или шестнадцатеричным представлением целого числа n.

Модули **math** и **cmath.** В этих модулях собраны математические функции для действительных и комплексных аргументов. Это те же функции, что используются в языке C. Ниже приведены некоторые функции модуля math..

acos(z) арккосинус z
asin(z) арксинус z
atan(z) арктангенс z
ceil(x) наименьшее целое, большее или равное x
cos(z) косинус z
cosh(x) гиперболический косинус x e константа e
exp(z) экспонента (то есть, e**z)
fabs(x) абсолютное значение x
floor(x) наибольшее целое, меньшее или равное x
fmod(x,y) остаток от деления x на y
log10(z) десятичный логарифм z

Например, без вызова модуля math, данная строка кода вызовет ошибку:

```
# код, который не работает
pi

# при импортировании модуля math:
import math
math.pi

# нужно не забывать добавлять имя модуля при вызове функции
math.cos(math.pi)
```

Следующие примеры показывают, как легко использовать Python для решения математических задач.

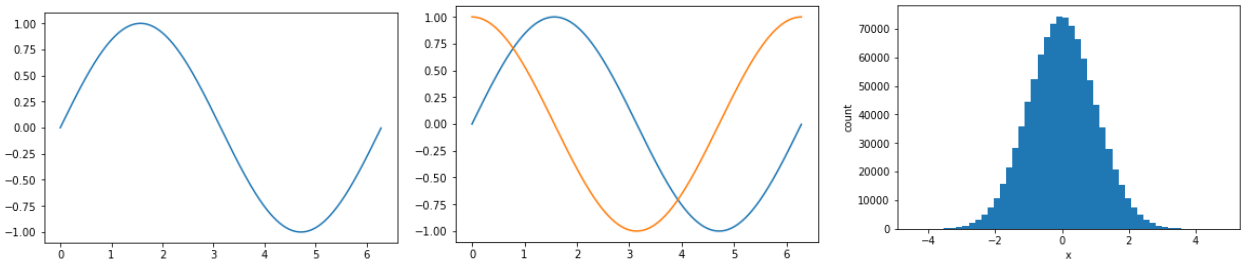
```
import math
import numpy
import matplotlib.pyplot as plt # используем сокращение plt
from mpl_toolkits import mplot3d
```

```
xvals = numpy.arange(0, 2*math.pi, 0.01) # определяем координату x
yvals = numpy.sin(xvals) # определяем координату y как функцию от переменной x
plt.plot(xvals, yvals) # создаем график
plt.show() # вызов графика функции, рис. 9(а)
```

Построение нескольких графиков:

```
xvals = numpy.arange(0, 2*math.pi, 0.01)
yvals = numpy.sin(xvals)
plt.plot(xvals, yvals)
yvals2 = numpy.cos(xvals) # добавляем новую переменную y2
plt.plot(xvals, yvals2) # добавляем на график вторую функцию
plt.show()# вызов графика функции, рис. 9(б)

# генерирование данных нормального распределения и построение гистограммы
plt.hist(numpy.random.randn(50000), bins=30)
plt.xlabel('x')
plt.ylabel('count')
plt.show()# вызов графика функции, рис. 9(в)
```



(а) (б) (в)

Рис. 9 Построение графиков в Python. (а) график функции *sin(x)*, (б) построение нескольких графиков, (в) генерирование данных нормального распределения.

Разберем пример построения более сложной функции:

$$f(x,y) = (1 - 2x + x^5 - y^3)(\exp(-x^2 - y^2))$$

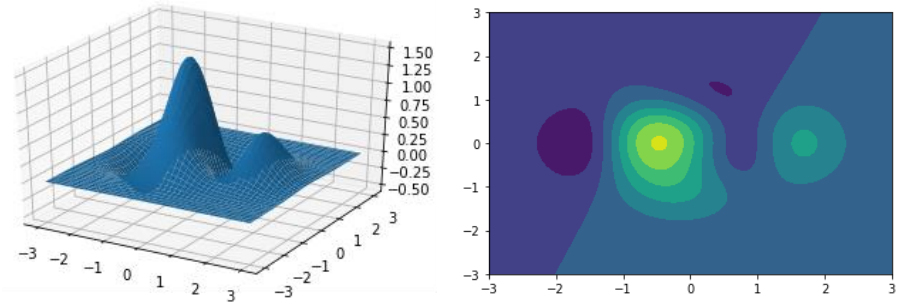
```
# определение функции в Python
def   f(x,y):
   return(1-2*x+x**5-y**3)*numpy.exp(-x**2-y**2)
```

Слово ***def*** в первой строке определяет, что далее будет определена функция. Первая строка является заголовком функции, в которой размещается имя функции и списки входных и выходных аргументов. В примере, приведенном выше, имя функции f, функция включает два аргумента x и y.

```
# создание сетки по оси x, y
n = 256
x = numpy.linspace(-3,3,n)
y = numpy.linspace(-3,3,n)
X,Y = numpy.meshgrid(x,y)
Z = f(X,Y)
ax = plt.axes(projection='3d')
```

ax.plot_surface(X,Y,Z)
plt.show() # построение графика рис. 10(а)

plt.contourf(X, Y, f(X,Y))
plt.show() # построение графика рис. 10(б)



(а) (б)

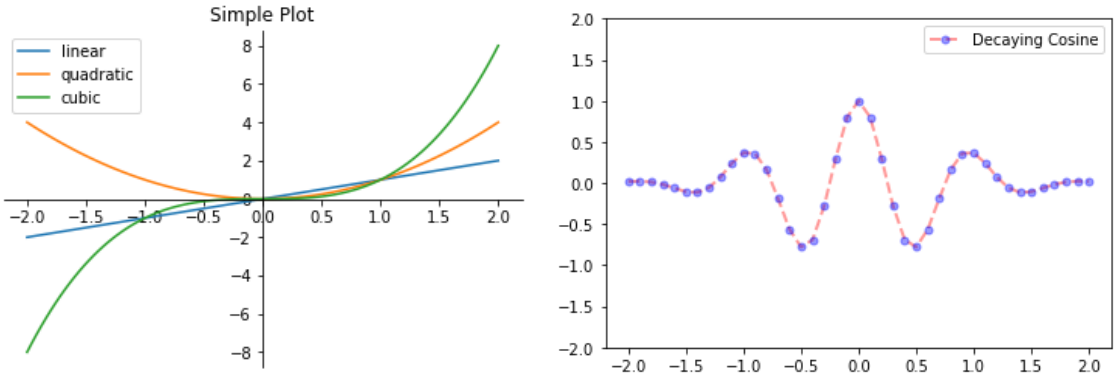
Рис. 10 Построение графиков сложной функции в Python.

Следующий код создает простой график линейного, квадратичного и кубического монома.

import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-2, 2, 100)
plt.plot(x, x, label='linear')
*plt.plot(x, x**2, label='quadratic')*
*plt.plot(x, x**3, label='cubic')*
plt.xlabel('x')
plt.ylabel('y')
plt.title("Simple Plot")
plt.legend()
plt.show()
 # работа с осями
ax = plt.gca()
 # сделаем верхнюю и правую ось невидимой
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
 # переместим вертикальную ось в ПОЗИЦИЮ y=0
ax.spines['bottom'].set_position(('data',0))
 # переместим горизонтальную ось в позицию x=0
ax.spines['left'].set_position(('data',0))
 # повторим построение графиков из предыдущего примера
x = np.linspace(-2, 2, 100)
plt.plot(x, x, label='linear')
*plt.plot(x, x**2, label='quadratic')*
*plt.plot(x, x**3, label='cubic')*
plt.title("Simple Plot")
plt.legend()
plt.show()

Другой пример:
x = np.linspace(-2,2,41)

```
y = np.exp(-x**2) * np.cos(2*np.pi*x)
plt.plot(x,y,alpha=0.4,label='Decaying Cosine',
        color='red',linestyle='dashed',linewidth=2,
        marker='o',markersize=5,markerfacecolor='blue',
        markeredgecolor='blue')
plt.ylim([-2,2])
plt.legend()
plt.show()
```



(а) (б)

Рис. 11 Построение графиков в Python: (а) представление линейного, квадратичного и кубического монома; (б) функция затухающегося косинуса.

Импортируем основные модули и посчитаем значение экспоненты $\exp(i\pi)$

```
import numpy as np
np.exp(np.pi*1j)
```

Рассмотрим функцию $f(x) = 1/(1-x)$. Данную функцию можно разложить в ряд Тейлора:

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + x^3 + \cdots$$

Данный ряд сходится при $[x] < 1$.

Построим графики $f(x)$ в зависимости от числа слагаемых

```
import matplotlib.pyplot as plt
x = np.linspace(-1.1, 1.1, 100)
plt.plot(x,1+x+x**2+x**3+x**4,label='degree 4')
plt.plot(x,1+x+x**2+x**3+x**4+x**5,label='degree 5')
plt.plot(x,1+x+x**2+x**3+x**4+x**5+x**6,label='degree 6')
plt.plot(x,1/(1-x),color='black',label='1/(1-x)')
plt.ylim([-2,6])
plt.xlabel('x')
plt.ylabel('y')
plt.title('Taylor polynomial approximations to 1/(1-x)')
plt.legend()
plt.show()
```

Вычисление факториала k! производится следующим образом:

import math
fac=math.factorial
print(fac(10))
print(fac(20))

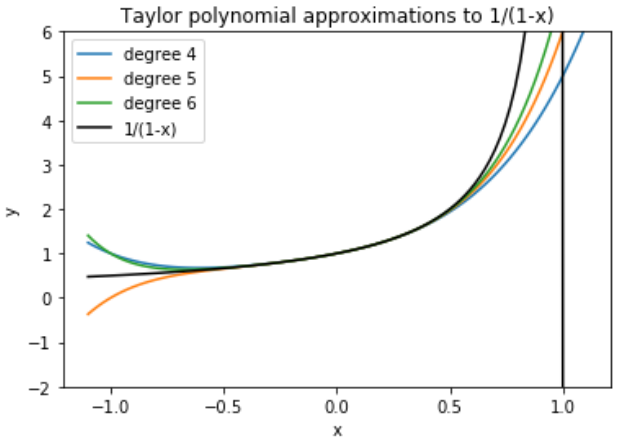


Рис. 12 Построение графиков в Python: разложение функции в ряд Тейлора.

2.1. Библиотека для анализа данных Pandas

Pandas – программная библиотека на языке Python для обработки и анализа данных. Работа в данной библиотеке строится с данными поверх библиотеки NumPy. Данная библиотека предоставляет специальные структуры данных и операции для манипулирования числовыми таблицами и временными рядами. Название библиотеки происходит от эконометрического термина «панельные данные», используемого для описания многомерных структурированных наборов информации, [6]. Добавление библиотек *numpy* и *pandas* осуществляется командами:

import numpy as np
import pandas as pd

Главными структурами данных библиотеки Pandas являются DataFrame и Series. Структура (объект) Series представляет собой объект, похожий на одномерный массив, но отличительной его чертой является наличие ассоциированных меток. Создадим объект Series путем передачи списка значений:

s = pd.Series([1, 2, 7, np.nan, 4, 9])

Создадим объект DataFrame путем передачи массива NumPy:

dates = pd.date_range('20200101', periods=8)

Мы можем указать таблицу значений:

df = pd.DataFrame(np.random.randn(8, 4), index=dates, columns=list('ABCD'))

Просмотр верхних и нижних значений можно осуществить с помощью функций *df.head()* и *df.tail()*.

Следующие три команды выдают соответственно информацию о значениях данных в колонках, характеристику строк и краткую статистическую информацию о таблице.

df.columns
df.index
df.describe()

Транспонирование таблицы данных можно осуществить с помощью команды *df.T*. Сортировку, например, колонки ‘B’ можно реализовать с помощью *df.sort_values(by='B')*.

Более полную информацию о библиотеке Pandas и краткое руководство по ее использованию можно посмотреть, например, здесь [7].

2.2. Задания

1. Построить график функции $y=x^5-x^4+x^3+1$
2. Построить график y=tan(x) на интервале от -4π до 4π.
3. Построить график окружности с радиусом R=7.
4. Вывести таблицу значений функции $y(x) = \frac{5 \cdot cos^2(x)}{3+\sin(2x)} + 4 \cdot e^{-x}$. Построить график функции на отрезке [0, 1].
5. Построить график функции: $f(x) = exp(x)$. Разложение данной функции в ряд Тейлора записывается следующим образом:

$$f(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Построить графики функции при разложении функции в ряд Тейлора для 5, 6 и 7 степени. (а) Вычислить значения функции при x = 1,5, и сравнить результат с exp (1.5). (а) Вычислить значения функции при x = 1+i, и сравнить результат с exp (1+i). (в) реализовать предыдущие пункты через цикл *for*.

6. Реализовать разложение в ряд функции синуса $f(x) = \sin(x)$ используя цикл *for*. Разложение данной функции в ряд Тейлора записывается следующим образом:

$$sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Построить графики функции при разложении функции в ряд Тейлора для 1, 3 и 7 степени.

7. С помощью библиотеки Pandas реализовать: (а) изменение данных столбца DataFrame по условию, (б) изменить позиции столбцов объекта, (в) заменить пропущенные значения в таблице средними значением.

ВСТРОЕННЫЕ ФУНКЦИИ В PYTHON

Python имеет полезные встроенные функции для работы с последовательностями. Ниже приведена таблица с полезными функциями:

Функция	Описание функции
print(object)	Выводит значения на экран
type(object)	Возвращает тип объекта.
abs(x)	Возвращает абсолютное значение числа x
int(x)	Целое число, построенное из числа x, усекая десятичную
len(sequence)	Возвращает длину последовательности
sum(sequence)	Возвращает сумму последовательности
max(sequence)	Возвращает максимум последовательности
min(sequence)	Возвращает минимум последовательности
range(a,b,step)	Возвращает объект от a до b с шагом step
list(sequence)	Возвращает список элементов из последовательности
sorted(sequence)	Возвращает сортированный список элементов
reversed(sequence)	Возвращает обратный отсортированный список элементов
enumerate(sequence)	Возвращает пронумерованный объект, сформированный из последовательности.
zip(a,b)	Объединяет последовательность a и b

Пример использования функции **int()**:

```
print(int(-1.414))
print(int(1.414))
```

Пример использования функции **zip()**:

```
list_1=[2,5,4,2]
list_b=[43,47,22,-34]
# print(zip(list_1,list_b))
print(list(zip(list_1,list_b)))
```

Пример использования функции **enumerate()**:

```
print(list_1)
range_a=range(len(list_1))
print(list(zip(range_a,list_1)))
print(list(enumerate(list_1)))
# print(enumerate(list_1))
```

Определим функцию для расчета среднего значения:

```
def average(x):
    "Compute the average of the values in the sequence x."
    # compute the sum of the sequence entries
    x_sum=sum(x)
    # compute the number of the sequence entries
    x_len=len(x)
    # return the quotient
```

```
    return x_sum/x_len
```

Вызов функции average:
```
average([1,2,3,4,5,6,7,8])
```

Строка документации **любой функции** доступна с помощью знака вопроса: ?

```
?average
```

Использование тройных кавычек *"""позволяет закомментировать несколько строк подряд"""*.

```
def average(x):
    """Compute the average of the values in the sequence x.
    Parameters
        x: any iterable with numerical values
    Returns:
        the average value of all the values in the iterable
    Examples
        >>> average([1,2,3])
        2.0
    """
    x_sum=sum(x)
    x_len=len(x)
    return x_sum/x_len
```

Рассмотри пример функции для расчёта производной, которая определяется согласно формуле:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```
def g(f,x,h):
    " Вычислить разностное отношение для функции f в точках x и x + h "
    return (f(x+h)-f(x))/h

print(g(np.sin,np.pi/4,0.001))
print(g(np.sin,np.pi/4,0.00001))
print(g(np.sin,np.pi/4,0.0000001))
print(g(np.sin,np.pi/4,0.000000001))
print(0.5**0.5)
```

3.1. Задания

1. Найти производную функции $f(x) = x^2$ в точке x=1, с точностью $h = 10^{-n}$, где n=1,2,3,…10.

2. Для последовательности x=[1 1 5000] с помощью оператора def определить функции:

(а) среднего значения;

(б) гармонического среднего, определяемого по формуле:

$$f(x) = \frac{n}{\sum_{k=1}^{n} \frac{1}{x_k}}$$

(в) функцию геометрического среднего, определяемого по формуле:

$$g(x) = \sqrt[n]{\prod_{k=1}^{n} x_k}$$

Сравнить между собой полученный результат.
Замечание:
- длина вектора x может быть определена командой len(x);
- суммирование элементов осуществляется командой sum(x);
- произведение элементов осуществляется командой prod(x).

ОПЕРАТОРЫ УСЛОВИЯ И СРАВНЕНИЯ В PYTHON

В этой главе мы познакомимся с операторами условия и сравнения в языке Python.

Определим функцию $f(x)$, которая принимает значения равного 1, если x лежит в интервале от 0 до 1 и приобретает значение, равное 0, для любого другого случая:

$$f(x) = \begin{cases} 1, \text{если } x \in (0,1) \\ 0, \text{если } x \notin (0,1) \end{cases}$$

На языке Python данную функцию можно определить следующим образом:

```
def f(x):
    if 0<x and x<1:
        return 1
    else:
        return 0
```

Построим график функции, которую мы только что написали:

```
from matplotlib import pyplot as plt
x=[-2+n/100 for n in range(500)]
y=[f(t) for t in x]
plt.plot(x,y)
plt.show()
```

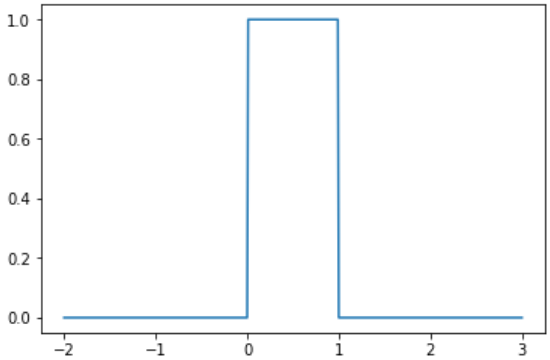


Рис. 13 Построение графика прямоугольного импульса в Python.

При использовании сравнения и логических операторов подразумевается использование логических типов, которые могут принимать значения True и False. В Python есть восемь операций сравнения:

Операция	Значение
<	строго меньше чем
<=	меньше или равно
>	строго больше чем
>=	больше или равно
==	равно
!=	не равно
is	идентичность объекта
is not	отрицание идентичности объекта

Для численных сравнений достаточно сосредоточиться на первых шести из них. (Идентификация объекта связана с тем, как данные хранятся в Python.)

```
print(0<=0)
print(0>0)
print(0>=0)
print(0==0)
print(0!=0)
print(0==1)
print(0!=1)
```

Булевы операторы **and**, **or**, и **not** могут быть использованы для объединения логических операций.

```
print(True and False)
print(True or False)
print(not True)
```

Рассмотрим пример программирования сложной функции $g(x)$:

$$g(x) = \begin{cases} 1+x & \text{для } x < -1 \\ \sqrt{-x(1+x)} & \text{для } -1 \le x < 0 \\ -\sqrt{x(1-x)} & \text{для } 0 \le x < 1 \\ x-1 & \text{для } x \ge 1 \end{cases},$$

Код на языке Python для этой функции выглядит следующим образом:

```
def g(x):
    if x<-1:                #x < -1
        y=x+1
    elif x<0:               # -1 <= x <  0
        y=(-x*(1+x))**(1/2)
    elif x<1:               #  0 <= x <  1
        y=-(x*(1-x))**(1/2)
    else:                   #  1 <= x
        y=x-1
    return y

from matplotlib import pyplot as plt
x=[-2+n/100 for n in range(400)]
y=[g(t) for t in x]
plt.plot(x,y)
plt.show()
```

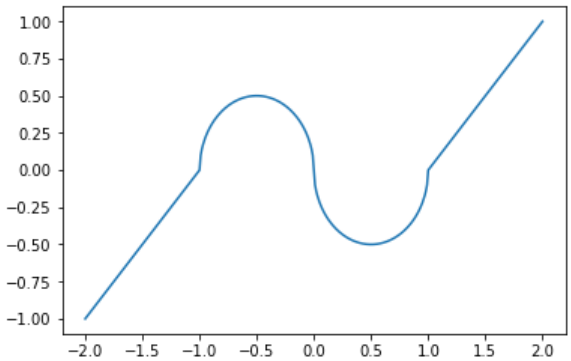


Рис. 14 Построение графика сложной функции в Python.

Цикл **for** выполняет блок кода несколько раз с некоторыми параметрами, обновляемыми каждый раз в цикле.

```
iterable=['some','iterable','such','as','a','list']
for item in iterable:
    print(item)
```

Цикл начинается с оператора **for**, ключевыми моментами использования данного цикла являются:
- слова **for** и **in** являются ключевыми.
- здесь *iterable* является объектом последовательности, таким же как список (list), диапазоном (range) или кортежем (tuple).
- переменная *item* переменная, которая принимает каждый раз значение из списка *iterable.*
- в конце строки с **for** стоит двоеточие.

Еще несколько примеров:

```
for x in [1,2,3,4]:
    print(x**2)

my_lst=[x**2 for x in [1,2,3,4]]
print(my_lst)
```

Выходные данные этого кода содержат квадраты всех элементов в списке [1,2,3,4]. Когда вы запускаете этот код, выходные данные появляются сразу, но здесь скрыто то, что выходные данные фактически создаются последовательно один за другим.

```
my_lst=[]
for x in [1,2,3,4]:
    my_lst.append(x**2)
print(my_lst)
```

Полезно добавить несколько команд print() для отслеживания значений переменных:

```
my_lst=[]
for x in [1,2,3,4]:
    my_lst.append(x**2)
    print(my_lst)
```

Цикл **for** полезен, если мы точно знаем, как часто мы хотим выполнить блок кода, поскольку мы обычно используем его с заданной итерацией фиксированной длины. Если мы не знаем, сколько раз мы хотим выполнить блок кода, цикл **while** более полезен, поскольку он позволяет нам повторять блок кода, пока логическое утверждение остается верным.

```
x=1
while x<5:
    print(x**2)
    x=x+1
```

Списки

Теперь мы ввели несколько конструкций, которые позволяют нам создавать последовательности. Давайте сначала вспомним три способа, которые мы использовали ранее, когда обсуждали итерируемые элементы, а именно: написание списков напрямую, использование диапазонов и построение списков.

```
print([5,11,17,23,29,35])
print(list(range(5,36,6)))
print([5+6*n for n in range (6)])
```

```
my_list=[]
for n in range(6):
    new_item=5+6*n
    my_list.append(new_item)
print(my_list)
```

Как мы только что видели выше, это также может быть записано с использованием цикла while.

```
my_list=[]
n=0
while n<6:
    new_item=5+6*n
    my_list.append(new_item)
    n=n+1
print(my_list)
```

4.1. Задания

Одной из нерешённых проблем математики является гипотеза Коллатца (сиракýзская проблема), которая получила широкую известность благодаря простоте формулировки. Названа по имени немецкого математика Лотара Коллатца (англ.), сформулировавшего эту задачу 1 июля 1932 года [8]. Для объяснения сути гипотезы рассмотрим следующую последовательность чисел, называемую сиракýзской последовательностью. Берём любое натуральное число n. Если оно чётное, то делим его на 2, а если нечётное, то умножаем на 3 и прибавляем 1 (получаем 3n + 1). Над полученным числом выполняем те же самые действия, и так далее. Гипотеза Коллатца заключается в том, что какое бы начальное число n мы ни взяли, рано или поздно мы получим единицу [9].

Гипотеза Коллатца задается функцией:

$$f(x) = \begin{cases} 3x+1 & \text{если } x - \text{нечетное} \\ x/2 & \text{, если } x - \text{четное} \end{cases}$$

последовательность задаётся следующим образом: $x_0=a$, $x_0= f(a)$, …, $x_{n+1}=f(x_n)$ для $x\in\mathbb{N}$.

Выполните следующие упражнения:

(а) Используя цикл for, напишите функцию iterate_Collatz (a,N), которая создает список итераций $x_0=a$, x_1, …, x_N.

(б) Рассмотреть случаи a = 27, a = 231 и a = 703. Попробуйте найти число N, при котором ряд заканчивается. Сигналом к завершению последовательности должна быть последовательность […, 8, 4, 2, 1].

(в) Постройте график полученной последовательности, для заданного числа a = 231 и определенного из предыдущего пункта N.

РАБОТА С ИЗОБРАЖЕНИЯМИ В PYTHON

Для создания изображения нам необходимо подключить необходимые библиотеки. Рассмотрим следующий код, который определяет черный квадрат, размером 100x100 px:

```
# импортируем необходимые библиотеки
from PIL import Image
from IPython.core.display import display
# создаем изображение размером 100x100 px
img=Image.new("RGB", (100,100))
display(img)
```

Ниже приведен код, который отображает цветной прямоугольник, используя обозначения RGB (аббревиатура английских слов red, green, blue — красный, зелёный, синий)— аддитивная цветовая модель, как правило, описывающая способ кодирования цвета для цветовоспроизведения с помощью трёх цветов, которые принято называть основными. Выбор основных цветов обусловлен особенностями физиологии восприятия цвета сетчаткой человеческого глаза. В компьютерах для представления каждой из координат представляются в виде одного октета, значения которого обозначаются для удобства целыми числами от 0 до 255 включительно, где 0 — минимальная, а 255 — максимальная интенсивность.

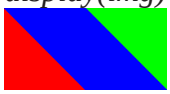
```
# определить некоторые цвета,
# используя обозначения RGB (красный, зеленый, синий)
# со значениями цвета от 0 до 255
colors =[(255, 0, 0), (0, 255, 0), (0, 0, 255)]
SizeX=150
SizeY=100
# создаем изораженне размером (SizeX, SizeY) px
img=Image.new("RGB", (SizeX,SizeY))
# закрашивем пиксели
for j in range(SizeY):
    for i in range(SizeX):
        if j<SizeY*(1/3):
            img.putpixel((i,j),colors[0])
        elif j>SizeY*(2/3):
            img.putpixel((i,j),colors[1])
        else:
            img.putpixel((i,j),colors[2])

# отображаем картинку на экран
display(img)
```

```
colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255)]
SizeX=200
SizeY=100
```

```
# создаем изображение размером (SizeX, SizeY) px
img=Image.new("RGB", (SizeX,SizeY))
# закрашивем пиксели
for i in range(SizeX):
    for j in range(SizeY):
        if i<j:
            img.putpixel((i,j),colors[0])
        elif i>j+100:
            img.putpixel((i,j),colors[1])
        else:
            img.putpixel((i,j),colors[2])

# отображаем картинку на экран
display(img)
```



5.1. Задания

1. Напишите код, для отображения красного круга радиуса R на зеленом фоне. Формула круга задается уравнением $(x-x_0)^2+(y-y_0)^2<r^2$. Определение красного и зеленого цвета: colors = [(255,0,0), (0,255,0)].
2. Напишите код, для отображения синего квадрата размером d на красном фоне. Формула квадрата задается уравнением $\max(|x - x_0|, |y-y_0|)<d$. Определение синего и красного цвета: colors = (255,0,0), (0,0,255)].
3. Напишите код для реализации: (а) движения красного круга внутри зеленого прямоугольника. (б) красный круг случайно меняет свое направление при столкновении с границей прямоугольника и движется прямолинейно до следующего столкновения со стенкой.

РАБОТА С ФУНКЦИЯМИ В PYTHON

Функция – это блок кода, который начинается с ключевого слова def, названия функции и двоеточия. Предположим, что функция имеет тело, которое является простым однострочным оператором:

```
def f(x):
    return x*x
```

Вместо этого выражения мы можем написать

```
f = lambda x: x*x
```

Название «лямбда» используется по историческим причинам. Это связано с формальной системой в математической логике, называемой лямбда-исчислением [10]. Преимущество лямбда-функции в том, что мы можем использовать ее, без определения имени для функции, поэтому лямбда-функции часто называют анонимными функциями:

```
(lambda x: x*x)(-4)
```

Рассмотрим функцию, которую мы определили для вычисления производной функции:

```
def g(f,x,h):
    "Compute the difference quotient for the function f at point x and x+h"
    return (f(x+h)-f(x))/h
```

Для того, чтобы использовать ее для функции f(x)=x^2, мы должны были определить новую функцию:

```
def f(x):
    return x**2
g(f,2,0.01) # вызов функции f(x)=x2 в точке x=2 с шагом h=0.01.
```

Лямбда-функция позволяет нам записать код для g(f,x,h) в более простом виде, без определения дополнительных функций:

```
g(lambda x: x**2,2,0.01)
```

Одной из основных задач математики является нахождение корней уравнений для функции f(x). Рассмотрим ключевые численные методы для решения этой задачи.

Метод деления отрезка пополам

Предположим, что у нас есть непрерывная вещественная функция f(x) на отрезке [a,b], и мы хотим найти решения уравнения f(x)=0. Согласно теореме о промежуточном значении [11] если функция непрерывна на некотором отрезке [a,b] и на концах этого отрезка принимает значения противоположных знаков, то существует точка на отрезке [a,b], в которой она равна нулю. Теперь, когда мы знаем, при каких условиях существует решение, как мы можем его вычислить? Мы просто делим интервал [a,b] на две равные части и применяем теорему о промежуточном значении, чтобы решить, в каком из этих меньших интервалов лежит корень. Далее мы повторяем деление пополам интервала, пока он не станет достаточно маленьким.

Алгоритм поиска корней уравнения методом деления отрезка пополам:

1. Начинаем с интервала $[a,b]$ такого, что $f(a)f(b)<0$, что эквивалентно, тому, что $f(a)$ и $f(b)$, имеющих противоположный знак. (В частности, (a) и $f(b)$ должны быть не равны нулю, иначе корни уже найдены).
2. Вычислить среднюю точку $m=(a+b)/2$. (Если (m) равно нулю, то решение райдено)
3. Определите, в каком подинтервале f изменяется знак:

(а) если $f(a)f(m)<0$, тогда выбираем следующий интервал $[a,m]$, т.е заменяем правую границу на m, используя $b=m$.

(б) если $f(m)f(b)<0$, тогда выбираем следующий интервал $[m,\ b]$, т.е заменяем правую границу на m, используя $a=m$.

4. Повторяем шаги (2) и (3), пока интервал $[a,b]$ не станет достаточно малым.
5. Находим точку m, как приближение корня.

Замечание:

В приведенном выше алгоритме в пункте (4) необходимо привести пояснения. Мы знаем, что исследуемый нами интервал $[a,b]$ каждый раз делится пополам при каждой итерации, таким образом при размер интервала $b\text{-}a$ каждый раз уменьшается на 2^N, тогда ошибка между корнем уравнения x и определенной средней m точкой при N итераций составит:

$$|m-x|<\frac{b-a}{2^N}$$

Таким образом, если мы хотим, чтобы ошибка была меньше некоторого значения ε, следовательно мы должны сделать $\frac{b-a}{2^N}<\varepsilon$ итераций, откуда $N>\frac{1}{log2}log_2\left(\frac{b-a}{\varepsilon}\right)$.

Код на Python

```
def bisection(f,a,b,N):
    for n in range(N):
        m=(a+b)/2
        if f(a)*f(m)<0:
            b=m
        elif f(m)*f(b)<0:
            a=m
    return (a+b)/2
```

Можно рассмотреть более подробный код для поиска корня уравнения методом деления пополам:

```
def bisection(f,a,b,N):
    # step 0. check for the obvious
    if f(a)==0:
        print ("Found exact solution.")
        return a
    if f(b)==0:
        print ("Found exact solution.")
        return b
    # step 1. ensure that f(a)f(b)<0
    if f(a)*f(b)>0: # (it cannot be zero because we already excluded this)
        print("Bisection method fails.")
        return None
    # now start the N-fold loop
    for n in range(N):
```

```
    # step 2. compute the midpoint
    m=(a+b)/2
    if f(m)==0:
       print ("Found exact solution.")
       return m
    # step 3.
    if f(a)*f(m)<0: # step 3.A.
       b=m
    elif f(m)*f(b)<0: # step 3.B.
       a=m
    else: # Logically, this should not happen.
        # But we're dealing with floating point operations. (*)
       print("Bisection method fails.")
       return None
  print("Computed approximate solution.")
  return (a+b)/2
```

Рассмотрим функцию $f(x)=(x-3)(x+1)$. Корни данной функции $x=3$ и $x=-1$. Воспользуемся функцией lambda:

```
print(bisection(lambda x:(x-3)*(x+1),-5,0,8))
print(bisection(lambda x:(x-3)*(x+1),0,5,8))
```

Можно рассмотреть более интересный пример, когда $f(x)=kx$, где k – некоторый коэффициент, постарайтесь объяснить полученный результат:

```
print(bisection(lambda x: x,-2,1,30))
print(bisection(lambda x: 1e-100*x,-2,1,30))
print(bisection(lambda x: 1e-200*x,-2,1,30))
```

Метод секущей

Метод секущих — модификация метода Ньютона, в котором производная (вычислять ее не всегда удобно) заменена на секущую.

Секущая — прямая, проходящая через две точки на графике функции. В данном методе процесс итераций состоит в том, что в качестве приближений корню уравнения принимаются последовательные значения точек пересечения секущей с осью абсцисс.

Положим, что у нас есть две точки, a и b, в которых значения функции равны соответственно f(a) и f(b). Тогда уравнение прямой, проходящей через эти точки, будет

$$\frac{y-f(x_1)}{f(x_1)-f(x_0)}=\frac{x-x_1}{x_1-x_0}$$

Для точки пересечения с осью абсцисс (y=0) получим уравнение:

$$x_k=x_1-\frac{x_1-x_0}{f(x_1)-f(x_0)}f(x_1)=\frac{x_1f(x_0)-x_0f(x_1)}{f(x_0)-f(x_1)}$$

Это и есть итерационная формула, которую можно переписать в виде:

$$x_{n+1}=\frac{x_nf(x_{n-1})-x_{n-1}f(x_n)}{f(x_{n-1})-f(x_n)}$$

В качестве критерия для завершения алгоритма выбирают: $f(x_n)<\varepsilon$ или $\lceil x_n-x_{n-1}\rceil<\varepsilon$.

Алгоритм поиска корней уравнения методом секущих:

1. Начинаем с интервала $[a,b]$.
2. Пусть $x_1=a$, $x_0=b$.
3. Для n от 0 до N-1:

$$x_{n+1} = \frac{x_n f(x_{n-1}) - x_{n-1} f(x_n)}{f(x_{n-1}) - f(x_n)}$$

4. Вывести x_N

Код на Python

```
def secant(f,a,b,N):
   x_new,x_old=a,b
   for n in range(1,N):
      x_new,x_old=(x_new*f(x_old)-x_old*f(x_new))/(f(x_old)-f(x_new)),x_new
   print("Computed approximate solution.")
   return x_new
print(secant(lambda x:(x-3)*(x+1),-5,0,8))
print(secant(lambda x:(x-3)*(x+1),0,5,8))
```

Но теперь второй вызов функции не сходится к корню x=3 в интервале (0,10). Это показывает слабость метода секущих. Попробуем поменять интервал:

```
print(secant(lambda x:(x-3)*(x+1),2,4,8))
```

6.1. Задания

1. Пусть исследуемый сигнал x(t) описывается периодическими прямоугольными импульсами с амплитудой A=2, периодом T=2. Напишите код, реализующий сигнал x(t) на интервале времени от -4 до 4.

Для определения прямоугольного импульса воспользоваться готовой функцией **signal.square,** которая определена в библиотеке **scipy.** Код для подключения библиотеки: ***from scipy import signal.*** Информацию о функции можно посмотреть с помощью команды: **?signal.square**

2. Написать программу для реализации метода Ньютона.

Метод Ньютона. Алгоритм Ньютона (также известный как метод касательных) — это итерационный численный метод нахождения корня (нуля) заданной функции. Метод был впервые предложен английским физиком, математиком и астрономом Исааком Ньютоном (1643—1727). Поиск решения осуществляется путём построения последовательных приближений и основан на принципах простой итерации.

Запишем уравнение касательной, проходящей через (a, f (a)):

$$y = f(a) + f'(x - a)$$

Мы хотим определить значение, x при котором y=0, тогда:

$$x = a - \frac{f(a)}{f'(a)}$$

Алгоритм поиска корней уравнения методом Ньютона:

1. Начинаем с точки a.
2. Пусть $x_0 = a$.
3. Для n от 0 до N-1:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

4. Вывести x_N

Найти корни уравнения:

(а) для функции $f(x)=x^2-2x-3$,

(б) комплексной функции $f(z)=z^2+1$.

РЕШЕНИЕ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЯ В PYTHON

Задача Коши для одного дифференциального уравнения n-го порядка состоит в нахождении функции, удовлетворяющей уравнения, [12]:

$$y^{(n)} = f\left(t, y, y', \dots, y^{(n-1)}\right)$$

И начальным условиям:

$$y(t_0) = y_1^0,\ y'(t_0) = y_2^0\ ,\dots,\ y^{(n-1)}(t_0) = y_n^0$$

Перед решением эта задача (уравнение или система уравнений) должна быть переписана в виде системы обыкновенных дифференциальных уравнение первого порядка

$$\frac{dy_1}{dt} = f_1(y_1, y_2, \dots, y_n, t)$$
$$\frac{dy_2}{dt} = f_1(y_1, y_2, \dots, y_n, t)$$
$$\dots\dots\dots\dots\dots\dots\dots\dots$$
$$\frac{dy_n}{dt} = f_n(y_1, y_2, \dots, y_n, t)$$

С начальными условиями $y_1(t_0) = y_1^0,\ y_2(t_0) = y_2^0,\ \dots,\ y_n(t_0) = y_n^0$

В языке программирования Python в модуле scipy.integrate есть две функции *ode()* и *odeint()*, предназначенных для решения систем обыкновенных дифференциальных уравнений (ОДУ) первого порядка с начальными условиями в одной точке (задача Коши). Функция *ode()* более универсальная, а функция *odeint()* имеет более простой интерфейс и хорошо решает большинство задач.

Функция *odeint()* имеет три обязательных аргумента и имеет следующий вид:

odeint(func, y0, t[,args=(), ...])

Аргумент *func* - это имя Python функции двух переменных, первой из которых является список y=[y1,y2,…yn], а второй - имя независимой переменной. Функция *func* должна возвращать список из n – значений функций $f_i(y_1, y_2, \dots, y_n, t)$ при заданном значении независимого аргумента t. Фактически функция *func(y,t)* реализует вычисление правых частей ОДУ.

Аргумент *y0* функции *odeint()* является массивом (списокм) начальных значений y_1^0, y_2^0, y_n^0 при $t = t_0$.

Третий аргумент функции *odeint()* является массивом моментов времени, в которые необходимо получить решение задачи. При этом первый элемент этого массива рассматривается как t_0.

Функция *odeint()* возвращает массив размера *len(t)*len(y0)* и имеет достаточно опций для управления. Опция *rtol* (r_{tol} – относительная погрешность) и *atoll* (a_{tol} – абсолютная погрешность) определяет погрешность вычислений ε_i для каждого значения y_i по формуле $\|\varepsilon_i\| \le r_{tol}|y_i|+a_{tol}$. Они могут быть векторами или скалярами. По умолчанию $r_{tol} = a_{tol} = 1.49012^{-8}$.

Рассмотрим задачу Коши $y' = -2t^3 \cdot y + 4y$, с начальным условием $y(-1) = 0.01$

```
import  numpy as np
from scipy.integrate  import odeint
import matplotlib.pyplot as plt

def dydt(y, t):
   return -(2*t**3)*y+4*y

t = np.linspace(-1,3,50) # вектор моментов времени
```

```
y0 = 0.01 # начальное значение
y = odeint(dydt, y0, t) # решение уравнения
y = np.array(y).flatten() # преобразование массива
plt.plot(t, y,'-sg',linewidth=2) # построение графика
# Добавляем подписи к осям:
plt.xlabel('t, время')
plt.ylabel('Y')
plt.show()
```

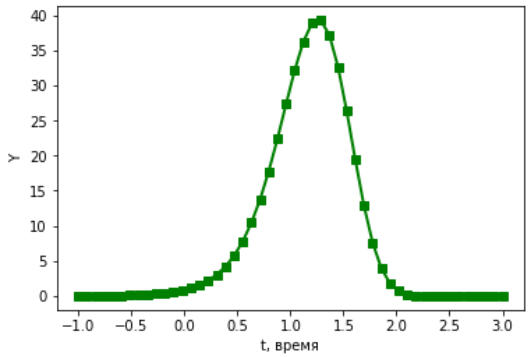


Рис. 15. Решение задачи Коши $y' = -2t^3 \cdot y + 4y$, с начальным условием $y(-1) = 0.01$

Рассмотрим следующий пример. Пусть необходимо решить дифференциальное уравнение $y'(x) + y = x$ с начальным условием $y(0) = 1$. Для этой задачи точное решение будет $y(x) = x - 1 + 2e^{-x}$, с которым мы можем сравнить наше решение.

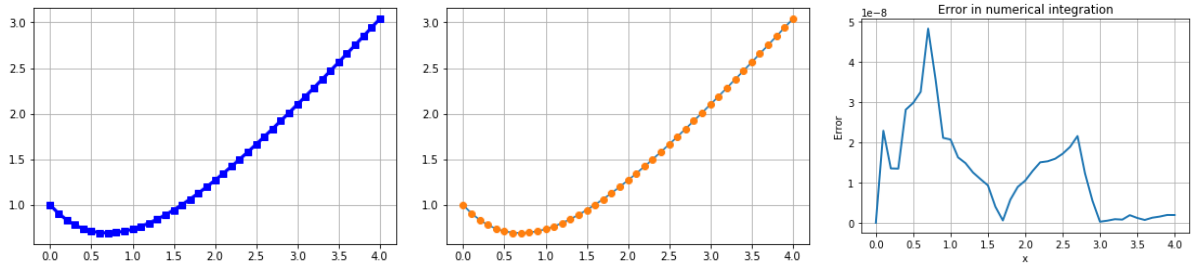
```
import  numpy as np
from scipy.integrate  import odeint
import matplotlib.pyplot as plt

def dydx(y, x): # функция правой части
    return x - y

x = np.linspace(0,4,41) # массив значений независимой переменной
y0 = 1.0 # начальное значение
y = odeint(dydx, y0, x) # решение уравнения
y = np.array(y).flatten() # преобразование массива
fig = plt.figure(facecolor='white') # построение графика
plt.plot(x, y,'-sb',linewidth=3)
ax = fig.gca()
ax.grid(True)

y_exact = x - 1 + 2*np.exp(-x) # Сравним графики «точного» и приближенного решений.
fig = plt.figure(facecolor='white')
ax = fig.gca()
ax.plot(x, y, x, y_exact, 'o');
ax.grid(True)

plt.show()
```



(а) (б) (в)

Рис. 16. Решение задачи дифференциального уравнения $y'(x) + y = x$ с начальным условием $y(0) = 1$:(а) численное решение; (б) точное решение; (в) ошибка между точным и приближенным решением.

Из рис. 16 видно, что абсолютная погрешность не превосходит $5 \cdot 10^{-8}$.

Исследуем решение задачи Коши

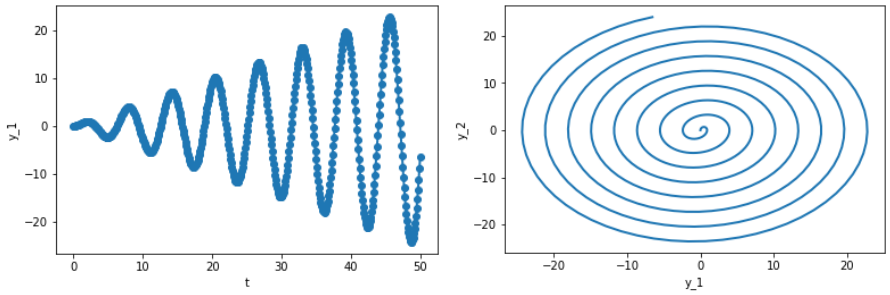
$$x'' + x = \cos(t),\ x(0) = 0,\ x'(0) = 0$$

Преобразуем данное уравнение 2-го порядка в систему уравнений 1-го порядка. Сделаем замену $x(t) = y_1(t),\ x'(t) = y_2(t)$, приходим к задаче:

$$\frac{dy_1}{dx} = y_2,\ \frac{dy_2}{dx} = -y_1(t) + \cos(t),\ y_1(0) = 0,\ y_2(0) = 0$$

```
import  numpy as np
from scipy.integrate  import odeint
import matplotlib.pyplot as plt
def f(y, t):
    y1, y2 = y # вводим имена искомых функций
    return [y2,-y1+np.cos(t)]

t = np.linspace(0,50,500)
y0 = [0, 0]
[y1,y2]=odeint(f, y0, t, full_output=False).T
fig = plt.figure(facecolor='white')
plt.plot(t,y1, '-o',linewidth=2) # график решения
plt.xlabel("t")
plt.ylabel("y_1")
fig = plt.figure(facecolor='white')
plt.plot(y1,y2,linewidth=2) # следующий рисунок слева
plt.xlabel("y_1")
plt.ylabel("y_2")
plt.show()
```



(а) (б)

Рис. 17. Решение задачи дифференциального уравнения $x'' + x = \cos(t)$, при начальных условиях $x(0) = 0,\ x'(0) = 0$

Исследуем двух видовую модель «хищник – жертва». Имеются два биологических вида, численностью в момент времени t соответственно x(t) и y(t). Особи первого вида являются пищей для особей второго вида (хищников). Численности популяций в начальный момент времени известны. Требуется определить численность видов в произвольный момент времени. Математической моделью задачи является система дифференциальных уравнений Лотки – Вольтерра:

$$\begin{cases} \frac{\mathrm{dx}}{\mathrm{dt}} = (a - by)x \\ \frac{\mathrm{dy}}{\mathrm{dt}} = (-c + dx)y \end{cases}$$

где x — количество жертв, y — количество хищников, t — время a – коэффициент рождаемости жертв, уменьшение жертв из-за хищников происходит за счет коэффициента b, c – коэффициент убыли хищников, сытые хищники способны к воспроизводству с коэффициентом d.

Проведем расчет численности популяций при $b = 1$, $c = 1$, $d = 1$ для трех значений параметра $a = 4{,}3{,}2$. Начальное значения положим $x(0) = 2$, $y(0) = 1$.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def f(y, t,params):
    y1, y2 = y
    a,b,c,d=params
    return [y1*(a-b*y2),y2*(-c+d*y1)]

t = np.linspace(0,7,300)
y0 = [1, 1]
fig = plt.figure(facecolor='white')

for a in range(4,1,-1):
    params=[a,1,1,1]
    st='a=%d b=%d c=%d d=%d' % tuple(params)
    [y1,y2]=odeint(f, y0, t,args=(params,), full_output=False).T
    plt.plot(y1,y2,linewidth=2, label=st)

plt.legend(fontsize=12)
plt.grid(True)
plt.xlabel("Число жертв")
plt.ylabel("Число хищников")
plt.show()
```

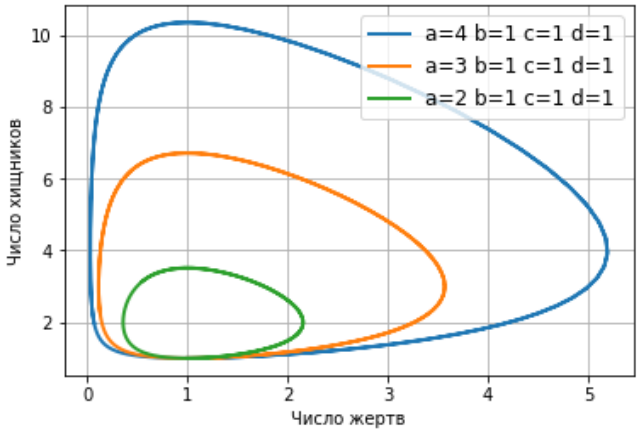


Рис. 18. Изменения числа хищников и их жертв в уравнении Лотки – Вольтерра.

7.1. Задания

1. Исследовать поведение математического маятника. Дифференциальное уравнение движения груза с массой равной единице имеет вид:

$$\varphi'' + k\varphi' + \omega^2 sin\varphi = 0$$

где $\varphi(t)$ – угол отклонения маятника от положения равновесия (нижнее положение), параметр k характеризует величину трения, $\omega^2 = g/l$, где g – ускорение свободного падения, l – длина маятника). Для начальных условий: $\varphi(0) = \varphi_0$, $\varphi'(0) = \varphi'_0$.

2. Решить задачу Коши, описывающую движение тела, брошенного с начальной скоростью v_0 под углом α к горизонту в предположении, что сопротивление воздуха пропорционально скорости. Уравнение движения имеет вид:

$$\boldsymbol{m\ddot{r}} = -\gamma \boldsymbol{v} - m\boldsymbol{g}$$

здесь $\boldsymbol{r}(t)$ радиус – вектор движищегося тела, $\boldsymbol{v} = \dot{\boldsymbol{r}}(t)$ – вектор скорости тела, γ – коэффициент сопротивления, $m\boldsymbol{g}$ – вектор силы веса тела массой m, g – ускорение свободного падения.

ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ В PYTHON

Многие прикладные задачи сводятся к определению интегралов. Однако аналитически вычислить удается немногие из них. Тогда интеграл можно найти численно. Для вычисления интегралов нам необходимо познакомиться с возможностью модулей SciPy в Python. Пакет SciPy является библиотекой математических процедур. Многие из них являются Python оболочками, обеспечивающими доступ к библиотекам, написанным на других языках программирования, в частности на языке C. Поскольку функции этих библиотек хранятся в откомпилированном виде, то эти процедуры обычно работают достаточно быстро. В этой части рассмотрим некоторые из функций пакета scipy.integrate, предназначенные для численного интегрирования. Приведем список наиболее часто используемых функций интегрирования из модуля:

Функция	**Описание**
quad	однократное численное интегрирование
dblquad	вычисление двойного интеграла
tplquad	вычисление тройного интеграла
nquad	вычисление n – кратного интеграла
cumtrapz	вычисление первообразной

Вычислим определенный интеграл:

$$Int = \int_{-2}^{2} \mathrm{e}^{-\mathrm{t}}\cos(\mathrm{t})\, dt$$

Первым шагом является подключение необходимы библиотек и создание функции, вычисляющей подынтегральное выражение. Функция f(x) должна содержать комбинацию стандартных математических функций, имена которых имеются в модуле numpy и scipy (или их подмодулях, например, в scipy.special). Она также может содержать вызовы функций пользователя.

```
from scipy. integrate import quad
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

def f(x):
   return np.exp( -x)*np.cos(x)
```

Для вычисления интеграла предназначена функция quad. Первым аргументом ей нужно передать ссылку на функцию f (имя функции), а вторым и третьим — нижний (a) и верхний (b) пределы интегрирования определяются командой . I=quad(f, a, b) Таким образом, вычислить интеграл можно следующим образом:

```
Int=quad(f ,  -2, 2)
print(I)
```

Напомним, что для вызова справки по функции можно воспользоваться командой:

```
?quad
```

Этот же интеграл можно вычислить используя лямбда-функцию:

```
g = lambda x: np.exp( -x)*np.cos(x)
Int2=quad(g, -2, 2)
print(Int2)
```

Функция f(x,...) может принимать несколько аргументов, для передачи подынтегральной функции дополнительных аргументов используется опция args, которой следует передавать кортеж с дополнительными аргументами. Рассмотрим вычисление интеграла, где a, b некоторые параметры:

$$Int = \int_0^1 (at^2 + bt)\, dt$$

```
def f(t, a, b):
   return a*t**2+b*t
```

Вычислим интеграл при $a = 3$, $b = 2$.

```
y,err  = quad(f , 0, 1, args=(3,2))
print("значение интеграла =",y,"ошибка интегрирования =",err)
```

Можно численно находить интеграл на бесконечном участке вещественной оси. Это значит, что верхний и/или нижний пределы интегрирования могут быть бесконечными. Для обозначения пределов ±∞ используется идентификатор numpy.inf. Например, вычислим интеграл в численном и аналитическом виде:

$$Int = \int_0^{+\infty} e^{-t^2}\, dt$$

В численном виде:

```
def f(x):
   return np.exp( -x**2)
I = quad(f , 0, np. inf)
print(I)
```

В аналитическом виде:

```
import sympy as smp
x=smp.symbols('x')
s1=smp. integrate(smp.exp( -x**2) ,(x,0,smp.oo))
print(s1)
print(s1.evalf())
```

Для вычисления повторных интегралов можно воспользоваться функцией dblquad. Функция dblquad имеет следующий синтаксис: scipy.integrate.dblquad(ffun, a, b, gfun, hfun[,...]). Здесь ffun является именем подынтегральной функции (двух переменных). При этом имя внутренней переменной интегрирования y при создании функции ffun должно быть указано первым. Аргументы a и b задают значения нижнего и верхнего пределов интегрирования по переменной x (пределы внешнего интеграла), gfun и hfun являются именами функций, определяющих нижний и верхний пределы интегрирования по переменной y (пределы внутреннего интеграла). Имеются также опции args, epsabs, epsrel, значение которых такое же, как и для функции quad.

Вычислим интеграл:

$$Int = \int_0^1 \int_0^x (x^2 + y^2)\, dxdy$$

```
from scipy. integrate import dblquad
f=lambda y,x: x**2+y**2
g=lambda x: 0
h=lambda x: x
I=dblquad(f, 0, 1, g, h)
print(I)
```

Кроме того, для вычисления n-кратных интегралов можно воспользоваться, приведем пример кода для вычисления интеграла:

$$Int = \int_0^\infty \int_1^\infty \left(\frac{e^{-xy}}{y^3}\right) dydx$$

```
from scipy. integrate import nquad
import numpy as np
def f(y, x) :
   return np.exp( -x*y) / y**3
I=nquad( f , [ [1, np. inf ] , [0, np. inf ] ])
print (I)
```

Обратите внимание на то, что порядок аргументов функции f должен соответствовать порядку, в котором передаются пределы интегрирования: от внутренней переменной к внешней. Внутреннему интегралу по аргументу y здесь соответствуют пределы [1,np.inf], а внешнему (по переменной x) – соответствуют пределы [0,np.inf]. Если хотя бы один из пределов интегрирования непостоянен, то оба элемента пары должны быть функциями. Вычислим интеграл, если p=1; q=2; r=3.

$$Int = \int_0^1 dx \int_0^{1-x} x^p y^q (1 - x - y)^r\, dy = \frac{p!\, q!\, r!}{(p + q + r + 2)!}$$

```
from scipy. integrate  import  nquad
import numpy as np
def f(x, y, p, q, r) :
   return x**p*y**q*(1-x-y)**r
def ybounds(x) :
     return [0, 1-x]
def xbounds( ):
     return [0, 1]
I=nquad( lambda y,x: f(x,y,1,2,3), [ybounds, xbounds])
print(I)
print(12/np.math. factorial(8) )    # точное значение
```

8.1. Задания

1. Вычислить численно интеграл:

$$Int = \int_0^1 dx \int_0^{\sqrt{1-x^2}} dy \int_0^{\sqrt{1-x^2-y^2}} \frac{x + y + z}{\sqrt{2x^2 + 4y^2 + 5z^2}} dz$$

3. Написать программу для вычисления интеграла от одиночного импульса, приведенного на рисунке с амплитудой A=2, см рис. 19.

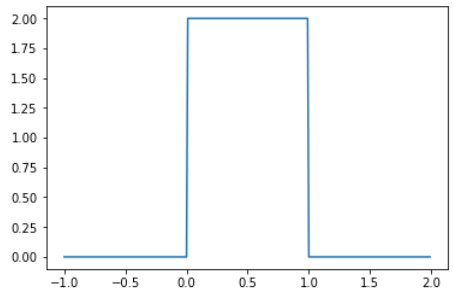


Рис. 19. График прямоугольного импульса с амплитудой A=2.

2. Пусть исследуемый сигнал x(t) описывается периодическими прямоугольными импульсами с амплитудой A=2, периодом T=2. Напишите код, реализующий сигнал x(t) на интервале времени от -4 до 4.

Периодический сигнал прямоугольных импульсов x(t) можно представить в виде ряда Фурье $x(t)^*$. Ограничимся конечным числом N слагаемых, тогда ряд Фурье можно записать в виде:

$$x(t)^* = \frac{a_0}{2} + \sum_{n=1}^{N}[a_n \cos(nwt) + b_n \sin(nwt)],$$

где коэффициенты ряда определяются по формулам:

$$a_0 = \frac{2}{T}\int_{t_0}^{t_0+T} \mathrm{x(t)}\, dt\,,$$

$$a_n = \frac{2}{T}\int_{t_0}^{t_0+T} \mathrm{x(t)} \cdot \cos(\mathrm{nwt})\, dt$$

$$b_n = \frac{2}{T}\int_{t_0}^{t_0+T} \mathrm{x(t)} \cdot \sin(\mathrm{nwt})\, dt$$

(а) Напишите программу, которая выведет на экран аппроксимированную функцию $\mathrm{x(t)}^*$,

(б) Постройте график погрешности приближения x(t) функцией $\mathrm{x(t)}^*$. Погрешность приближения можно определить по формуле $\varepsilon = \mathrm{x(t)} - \mathrm{x(t)}^*$.

Результат работы программы для N=10 слагаемых представлен на рис. 20.

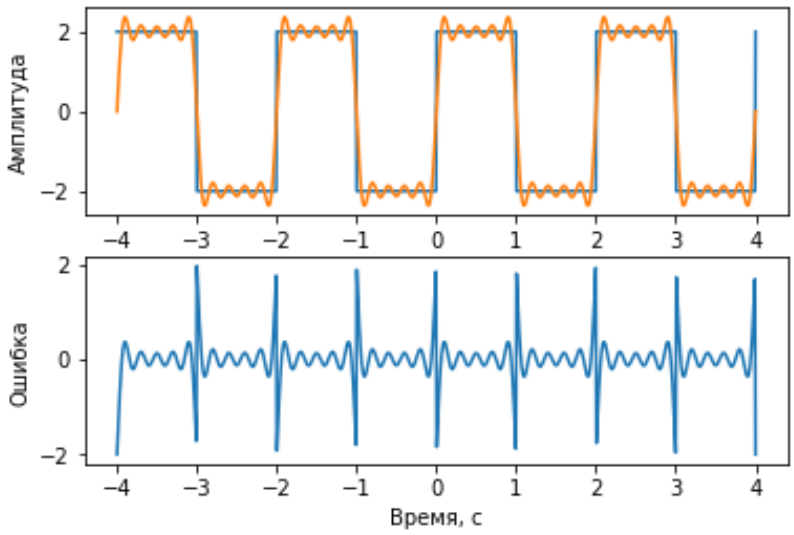


Рис. 20 Исходный сигнал x(t) аппроксимированный функцией x*(t), при N=10 и график погрешности приближение x(t) функцией $x(t)^*$.

ПРЕОБРАЗОВАНИЕ ФУРЬЕ В PYTHON

Данный раздел посвящен изучению дискретного (ДПФ) и быстрого преобразования Фурье (БПФ). Быстрое преобразование Фурье (БПФ) является одним из важнейших алгоритмов обработки сигналов и анализа данных [13].

9.1. Дискретное преобразование Фурье в Python.

Для задач цифровой обработки сигналов необходимо знать как дискретный сигнал $x(k)$, так и дискретный спектр сигнала $X(k)$, который можно сохранить в памяти цифрового устройства. Преобразование из $x(n) \to X(k)$ является переводом из временного пространства в частотное. Дискретный спектр сигнала $X(k)$ можно определить из прямого дискретного преобразования Фурье (ДПФ), которое вычисляется по формуле:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-i(2\pi kn/N)} \quad (1)$$

Обратное дискретное преобразование Фурье позволяет осуществить переход от дискретного спектра сигнала X_k к самому дискретному сигналу x_k по формуле:

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) \cdot e^{i(2\pi kn/N)} \quad (2)$$

Дискретное преобразование Фурье (ДПФ) может быть полезно как для исследования спектра мощности сигнала, так и для решения различных прикладных задач, в частности, при решении уравнения Шредингера из квантовой физики [14].

Python содержит множество стандартных инструментов вычисления ДПФ и БПФ. Библиотеки NumPy и SciPy имеют оболочки из библиотеки FFTPACK, которые находятся в подмодулях numpy.fft и scipy.fftpack` соответственно. (http://www.fftw.org/), (https://pypi.python.org/pypi).

Рассмотрим вычисление прямого дискретного преобразования Фурье (ДПФ). Из приведенного выше выражения для преобразования из $x(n) \to X(k)$, можно заметить, что оно эквивалентно линейной операции умножения матрицы M_{kn} на вектор $x(n)$:

$$\overrightarrow{X(k)} = M_{kn}\overrightarrow{x(n)},$$

где матрица определяется следующим образом:

$$M_{kn} = e^{-i(2\pi kn/N)}.$$

Обратное преобразование может быть реализовано похожим образом.

Рассмотрим код, который позволит вычислить ДПФ с использованием простого умножения матрицы на вектор:

```
import numpy as np
from scipy.fftpack import fft
import matplotlib.pyplot as plt
def DFT_slow(x):
    """Compute the discrete Fourier Transform of the 1D array x"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)
```

Воспользуемся стандартной функцией Python для быстрого преобразования Фурье fft() и сравним ее с реализованной функцией DFT_slow():

```
N = 600 # Количество отсчетов
fmax=800 # максимальная частота
T = 1.0/fmax #
f=100 # частота сигнала в Гц

x = np.linspace(0.0, N*T, N)
y = np.sin(f * 2.0*np.pi*x) #

yf = DFT_slow(y)
xf = np.linspace(0.0, fmax/2, N//2)
yff = fft(y)

fig = plt.figure(figsize=(8,8)) # размер полотна
plt.subplots_adjust(wspace=0.4, hspace=0.4) # отступ между графиками

plt.subplot(221)
plt.plot(xf, 2.0/N * np.abs(yf[0:N//2]))
plt.grid()
plt.xlabel('Частота, Гц');
plt.ylabel('Амплитуда');

plt.subplot(222)
plt.plot(xf, 2.0/N * np.abs(yff[0:N//2]))
plt.grid()
plt.xlabel('Частота, Гц')
plt.ylabel('Амплитуда')
```

Результат работы программы:

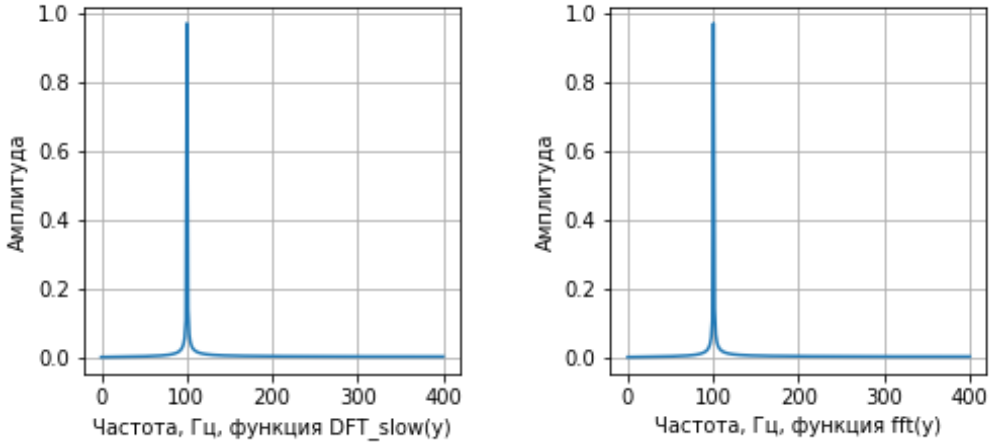


Рис. 21 Вычисление дискретного преобразования Фурье по функции DFT_slow() и встроенной функции fft(y).

Чтобы сравнить время выполнения реализованного алгоритма (ДПФ) и встроенного алгоритма быстрого дискретного преобразования Фурье (БДПФ) напишем код на Python:

```
%timeit DFT_slow(x)
%timeit np.fft.fft(x)
```

Из сравнения времени выполнения двух алгоритмов видно, что написанный алгоритм (ДПФ) более чем в 100 раз медленнее встроенного быстрого дискретного преобразования Фурье (БДПФ), что и следовало ожидать для такой упрощенной реализации.

Вычислительную сложность алгоритма дискретного преобразования Фурье (ДПФ) для вектора длины **N** можно оценить как $\mathbf{N^2}$, вычислительная сложность (БДПФ) оценивается как NlogN. Это означает, что для вектора из $N=10^6$ элементов БПФ завершится за 50 мс, в то время как алгоритм ДПФ займет около 20 часов!

Так как же для ДПФ добиться этого ускорения? Ответ заключается в использовании симметрии.

9.2. Быстрое преобразование Фурье в Python.

В данном разделе мы рассмотрим один из наиболее распространенных алгоритмов БПФ: БПФ по основанию два с прореживанием по времени для исходной временной последовательности x_n, для $n = 0,1,\dots,N-1$.

Приведем еще раз выражение для ДПФ:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot \exp(-i\frac{2\pi kn}{N}), \qquad k = 0,1,\dots,N-1 \tag{3}$$

Обозначим поворотные коэффициенты:

$$W_N^k = \exp\left(-i\frac{2\pi k}{N}\right), \qquad k = 0,1,\dots,N-1 \tag{4}$$

Тогда ДПФ примет упрощенный вид:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk}, \qquad k = 0,1,\dots,N-1 \tag{5}$$

В работе [13] было показано, что можно разделить вычисления ДПФ на две меньшие части. Проведем процедуру прореживания по времени, которая заключается в разделении исходной последовательности $x_1(m)$ для $n = 0,1,\dots,N-1$ на две последовательности длительности $x_0(m)$ и $x_1(m)$, где $m = 0,1,\dots,\frac{N}{2}-1$, Последовательность $x_0(m) = x(2m)$ содержит отсчеты с четными индексами, а последовательность $x_1(m) = x(2m+1)$ содержит отсчеты с нечетными индексами. Прореживание по времени для $N = 8$ наглядно представлено на рис. 22.

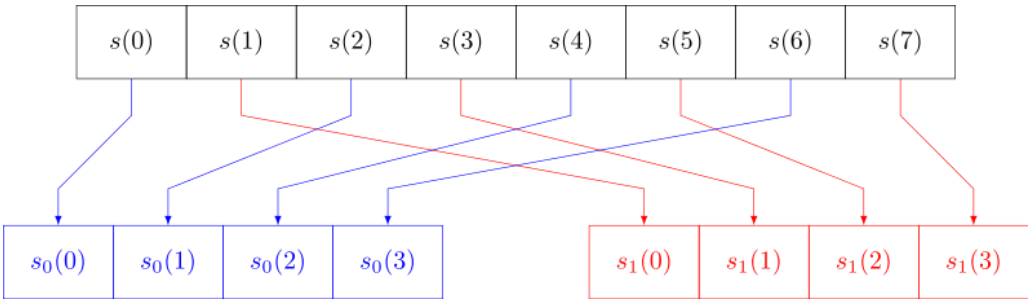


Рис. 22. Прореживание по времени для N=8.

Рассмотрим ДПФ прореженного по времени сигнала:

$$X(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_N^{2mk} + \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot W_N^{(2m+1)k} = \tag{6}$$

$$= \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_N^{2mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot W_N^{2mk}, \qquad k = 0,1,\dots,N-1$$

Рассмотрим поворотный коэффициент W_N^{2mk}:

$$W_N^{2mk} = \exp\left(-i\frac{2\pi}{N}2mk\right) = \exp\left(-i\frac{2\pi}{N/2}mk\right) = W_{N/2}^{mk} \tag{7}$$

Рассмотрим первую половину ДПФ $X(k)$ для индексов $k = 0,1,\dots,N/2-1$:

$$X(k) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_{\frac{N}{2}}^{mk} + W_N^k \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot W_{\frac{N}{2}}^{mk} = \\ = X_0(k) + W_N^k \cdot X_1(k), \qquad k = 0,1,\dots,N/2-1 \tag{8}$$

где:

$$X_0(k) = \sum_{m=0}^{\frac{N}{2}-1} x_0(m) \cdot W_{\frac{N}{2}}^{mk}, \qquad k = 0,1,\dots,\frac{N}{2}-1$$
$$X_1(k) = \sum_{m=0}^{N/2-1} x_1(m) \cdot W_{N/2}^{mk}, \qquad k = 0,1,\dots,N/2-1 \tag{9}$$

Проанализируем теперь вторую ДПФ $X(k+N/2)$ для индексов $k = 0,1,\dots,N/2-1$:

$$X(k+N/2) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_{\frac{N}{2}}^{m(k+\frac{N}{2})} + \\ + \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot W_{\frac{N}{2}}^{(2m+1)(k+\frac{N}{2})}, k = 0,1,\dots,N/2-1 \tag{10}$$

Рассмотрим поворотные коэффициенты:

$$W_N^{2m(k+\frac{N}{2})} = W_N^{2mk} W_N^{2m\frac{N}{2}} = W_{N/2}^{mk} W_N^{mN} = W_{N/2}^{mk}$$
$$W_N^{(2m+1)(k+\frac{N}{2})} = W_N^{2mk} W_N^{2m\frac{N}{2}} W_N^k W_N^{\frac{N}{2}} = W_{N/2}^{mk} W_N^{mN} = -W_N^k W_{N/2}^{mk} \tag{11}$$

Здесь $W_N^{mN} = \exp\left(-i\frac{2\pi}{N}mN\right) = 1$ и $W_N^{N/2} = \exp\left(-i\frac{2\pi}{N}mN/2\right) = -1$.

Тогда для второй части ДПФ $X(k+N/2)$ получим:

$$X\left(k+\frac{N}{2}\right) = \sum_{m=0}^{\frac{N}{2}-1} x(2m) \cdot W_{\frac{N}{2}}^{mk} - W_N^k \sum_{m=0}^{\frac{N}{2}-1} x(2m+1) \cdot W_{\frac{N}{2}}^{mk} = \\ = X_0(k) - W_N^k \cdot X_1(k), \qquad k = 0,1,\dots,N/2-1 \tag{12}$$

Используя выражение для первой и второй половин ДПФ, можно записать процедуру объединения как:

$$X(k) = X_0(k) + W_N^k \cdot X_1(k), \qquad k = 0,1,\dots,N/2-1$$
$$X(k+\frac{N}{2}) = X_0(k) - W_N^k \cdot X_1(k), \qquad k = 0,1,\dots,N/2-1 \tag{13}$$

где:

$$X_0(k) = \sum_{m=0}^{\frac{N}{2}-1} x_0(m) \cdot W_{\frac{N}{2}}^{mk}, \qquad k = 0,1,\dots,N/2-1 \tag{14}$$

$$X_1(k) = \sum_{m=0}^{N/2-1} x_1(m) \cdot W_{N/2}^{mk}, \qquad k = 0,1, \dots, N/2 - 1$$

Формулы (13) и (14) являются алгоритмом для реализации быстрого преобразования Фурье, в которых мы разделили исходное дискретное преобразование Фурье $X(k)$ на два слагаемых $X_0(k)$ и $X_1(k)$. Первое слагаемое $X_0(k)$ содержит отсчеты с четными индексами, второе слагаемое $X_1(k)$ содержит отсчеты с нечетными индексами. Каждое слагаемое состоит из **(N/2)*N** вычислений, в общей сложности $\mathbf{N^2}$. Основная хитрость быстрого преобразования Фурье заключается в использовании симметрии в каждых из этих слагаемых. Поскольку диапазон для k отсчетов, где $0 \leq k \leq N-1$, а диапазон для m отсчетов, $0 \leq m \leq M = \frac{N}{2} - 1$ из свойств симметрии видно, что необходимо выполнить только половину вычислений. Таким образом, общее число вычислений стало не $\mathbf{N^2}$, а $\mathbf{M^2}$, где $M = N/2$. Так как этот процесс итерационный, то данный метод приведет к уменьшению вычислительных затрат. В асимптотическом пределе этот рекурсивный подход для быстрого дискретного преобразования Фурье (БДПФ) имеет вычислительную сложность $Nlog(N)$.

9.3. Задания

1. На вход передатчика сигнала подается сигнал косинуса с частотами 50 и 150 Гц.

(а) Постройте дискретный спектр сигнала с помощью стандартной функции быстрого преобразования Фурье ***fft()*** и функции ***DFT_slow()***. Сравните время вычисления этих двух функций.

(б) С помощью функции быстрого обратного преобразования Фурье ***ifft()*** убедитесь, что форма сигнала не изменилась.

(в) После распространения сигнала в линии на приемнике был получен сигнал, искаженный белым шумом. Постройте дискретный спектр зашумленного сигнала на приемнике. Возьмите обратное преобразование Фурье и посмотрите, как поменялась форма принятого сигнала после распространения в линии.

Замечание: белый шум (Аддитивный белый гауссовский шум, [15]) можно добавить с помощью команды: *np.random.normal(0, 1, x.shape)*

2. Исследуемый сигнал x(t) описывается периодическими прямоугольными импульсами с амплитудой A=2, периодом T=2 на интервале времени от 0 до 4. Постройте дискретный спектр сигнала с помощью стандартной функции быстрого преобразования Фурье ***fft()*** и функции ***DFT_slow()***. Наложите шум на данный сигнал, посмотрите спектр зашумленного сигнала.

3. Реализовать код для определения быстрого дискретного преобразования Фурье (БДПФ). Убедиться, что функция написана верно, для этого рассмотреть сигнал косинуса с частотой 50 Гц и убедиться, что сигнал имеет одну выделенную частоту 50 Гц. Сравните время выполнения написанного кода для реализации (БДПФ) с встроенной функцией (БДПФ) ***fft().***

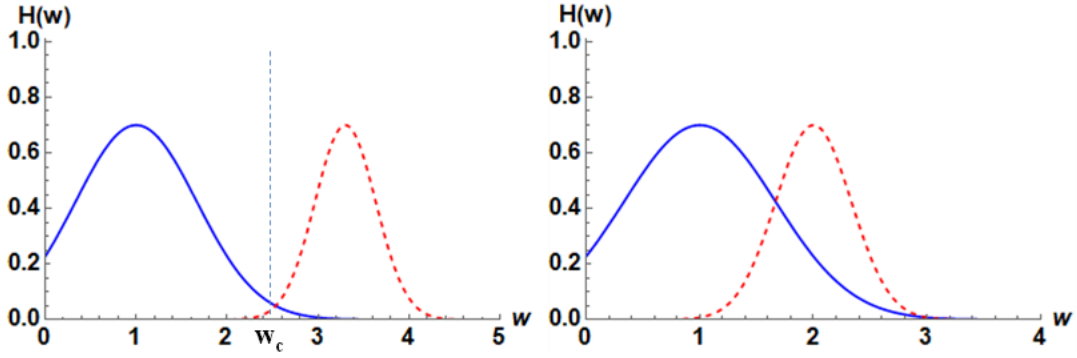
Замечание: при работе со срезами массивов могут быть полезны следующие свойства

```
col = 'abcdefgh'
# print(col[:])
# print(col[::-1])
print(col[::2])
print(col[1::2])
print(col[:2])
print(col[2:])
```

ФИЛЬТРАЦИЯ СИГНАЛОВ

10.1. Фильтр Баттерворта.

Фильтрация сигналов может быть представлена как процесс изменения частотного спектра в желаемом направлении. Этот процесс может привести к усилению или ослаблению частотных составляющих в определенном диапазоне. Разработка практических методов фильтрации базируется на принципе, что спектры полезного сигнала (1) и сигнала помехи (2) не перекрываются (см. рис. 23). С помощью фильтра, который пропускает все частотные составляющие в диапазоне $[0, \omega_c]$ и подавляет частотные составляющие в диапазоне (ω_c, ∞) довольно легко выделить полезный сигнал (1), рис. 23 (а). Задача фильтрации существенно усложняется, если спектры полезного сигнала и сигнала помехи перекрываются, рис. 23, (б). В этом случае ставят и решают задачу поиска оптимального фильтра.



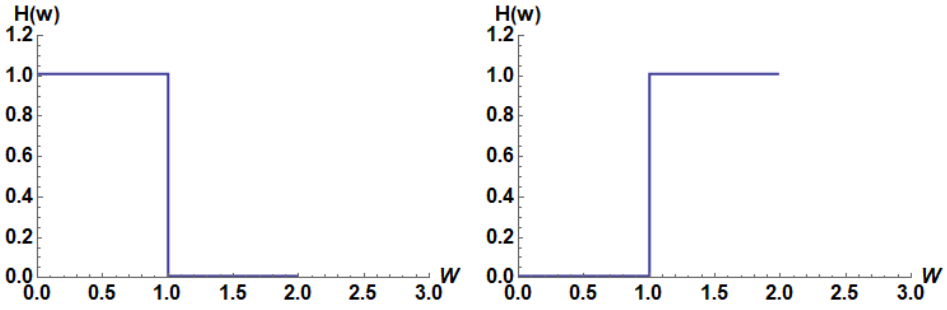
(а) (б)

Рис. 23. Амплитудные спектры полезного сигнала (1) и помехи (2):
(а) – с незначительным перекрытием, (б) – со значительным перекрытием.

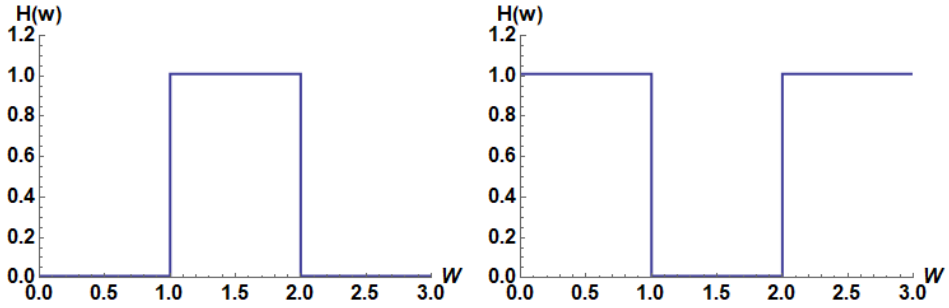
Полоса частот, в которой сигналы пропускаются (усиливаются) фильтром, называется **полосой пропускания**. Полоса частот, где сигналы подавляются (ослабляются) фильтром, называется **полосой задерживания**. Частоты, лежащие на границе полос пропускания и задерживания, называются **граничными частотами**. В зависимости от взаимного расположения полос пропускания и задерживания различают следующие типы фильтров:

- фильтры нижних частот (ФНЧ), (рис. 24, а);
- фильтры верхних частот (ФВЧ), (рис. 24, б);
- полосовые фильтры (ПФ), (рис. 24, в);
- заграждающие (режекторные) фильтры (ЗФ), (рис. 24, г).

Перечисленные выше типы фильтров широко применяются при обработке данных и сигналов. Поэтому их часто называют **базисными фильтрами**. В идеале базисные фильтры должны иметь амплитудно-частотные характеристики, представленные на рис. 24.



(а) (б)



(в) (г)

Рис. 24. Идеальные АЧХ базисных фильтров: (а) фильтр нижних частот (ФНЧ); (б) фильтр верхних частот (ФВЧ); (в) полосовой фильтр (ПФ); (г) заграждающий фильтр (ЗФ).

Модель фильтра как "черного ящика" представляется в виде линейного четырехполюсника с сосредоточенными параметрами, рис. 25. В частотной области основной характеристикой устройства (например, фильтра) является частотный коэффициент передачи $H(w)$.

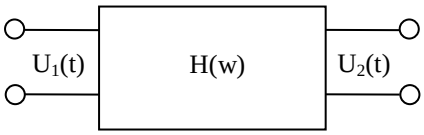


Рис. 25. Схема четырёхполюсника

Здесь $U_1(t)$ и $U_2(t)$ - входной и выходной сигнал, который зависит от времени. Используя преобразование Фурье получим спектр входного и выходного сигнала, $U_1(w)$ и $U_2(w)$.

$$U(w) = \int_{-\infty}^{+\infty} U(t)e^{-jwt}\,dt \tag{15}$$

Частотный комплексный коэффициент передачи $H(w)$ определяется как отношение спектра выходного сигнала к спектру входного сигнала.

$$H(w) = \frac{U_2(w)}{U_1(w)} \tag{16}$$

В общем случае частотный коэффициент передачи $H(w)$ есть комплексная функция $H(w) = A(w) + jB(w)$ или в показательной форме:

$$H(w) = |H(w)|e^{-j\varphi(w)} \tag{17}$$

где $|H(w)|$ – амплитудно-частотная характеристика (АЧХ) фильтра,

$\varphi(w) = \arg(H(w)) = atan\left(\frac{B(w)}{A(w)}\right)$ – фазочастотная характеристика (ФЧХ) фильтра, которая определяется как артангенс отношения мнимой части $H(w)$ к действительной части $H(w)$ или как аргумент комплексного коэффициента передачи $H(w)$.

Если заменить в преобразовании Фурье оператор jw на оператор (переменную) Лапласа $p = a + jw$, тогда преобразование Фурье примет вид (преобразование Лапласа):

$$U(p) = \int_{-\infty}^{+\infty} U(t)e^{-pt}\,dt \tag{18}$$

И передаточная функция, которая определяется отношением изображения $U_2(p)$ выходного сигнала к изображению $U_1(p)$ входного сигнала примет вид:

$$H(p) = \frac{U_2(p)}{U_1(p)} \tag{19}$$

Передаточная функция линейного четырехполюсника с постоянными параметрами может быть представлена в следующем виде:

$$H(p) = H_0 \frac{(p - z_1) \cdot (p - z_2) \cdot \ldots \cdot (p - z_m)}{(p - p_1) \cdot (p - p_2) \cdot \ldots \cdot (p - z_n)} \quad (20)$$

В общем виде полагают H_0 постоянной величиной и будем считать $H_0 = 1$; $z_1, \ldots, z_m$ и $p_1, \ldots, p_m$ - это нули и полюса передаточной функции, причем число полюсов n не должно превышать число нулей m.

Для устойчивости цепи полюсы $p_1, \ldots, p_m$ должны располагаться в левой полуплоскости комплексной частоты p, образуя комплексно-сопряженные пары.

При анализе и синтезе устройств часто используют частотный коэффициент мощности $H_p(w)$. Под ним понимают квадрат модуля частотного коэффициента передачи (квадрат АЧХ) четырехполюсника:

$$H_p(w) = H(w)H_c(w) = H(w)H(-w) = (|H(w)|)^2 \quad (21)$$

В отличие от комплексного коэффициента передачи $H(w)$ функция $H_p(w)$ всегда вещественна и поэтому удобна для задания исходных данных. Если вместо переменной частоты w подставить переменную p то функция $H_p(w)$ будет аналитически продолжаться с мнимой оси jw на всю плоскость комплексный частот p. Квадрат модуля (квадрат АЧХ) четырехполюсника:

$$H_p(p) = H(p)H_c(p) = H(p)H(-p) \quad (22)$$

10.2. Фильтр Баттерворта нижних частот.

В данном разделе мы рассмотрим фильтр нижних частот (ФНЧ), идеальный обобщенную частотную характеристику которого можно записать так:

$$H(w) = \begin{cases} 1, |w| \le w_c \\ 0, |w| > w_c \end{cases} \quad (23)$$

Основными характеристиками фильтра нижних частот (ФНЧ) являются частотный коэффициент передачи напряжения $H(\mathrm{w})$ и граничная частота, называемая частотой среза фильтра $\mathrm{w_c}$. Основное назначение таких устройств - передавать с минимальным ослаблением на выход колебания, частоты которых не превосходят частоты среза $\mathrm{w_c}$. Колебания с частотами $\mathrm{w} > \mathrm{w_c}$ должны существенно ослабляться. За единицу измерения величины ослабления (затухания) Δ напряжения фильтром принят децибел (дБ). Оценка величины ослабления АЧХ устройства в децибелах имеет вид:

$$\Delta(w) = 20\log(H(w))$$

Рассмотрим пример фильтра ФНЧ на примере фильтра Баттерворта. Коэффициента **передачи мощности** (квадрат АЧХ) **фильтра Баттерворта** описывается выражением:

$$H^2(w, n) = \frac{1}{1 + (w/w_c)^{2n}} \quad (24)$$

где w_c – граничная частота, n – порядок фильтра. АЧХ фильтров Баттерворта различных порядков представлены на рис. 26. При $\mathrm{n} \to \infty$ АЧХ фильтра Баттерворта приближается к идеальной.

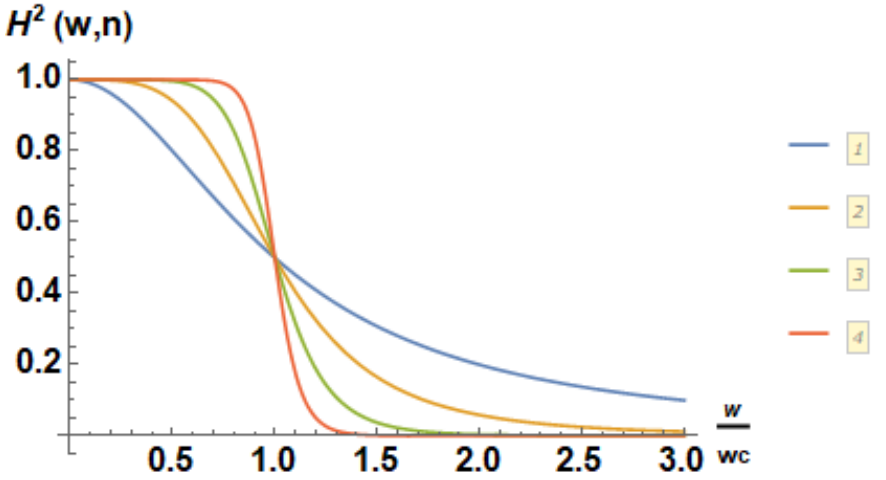


Рис. 26. Амплитудно-частотные характеристика фильтра Баттерворта: (1) для n=1, (2) для n=2, (3) для n=4, (4) для n=8.

Перепишем формулу для $H^2(w,n)$ в виде:

$$H^2(w,n) = \frac{w_c^{2n}}{w_c^{2n} + w^{2n}} \tag{25}$$

Тогда вместо текущей частоты w введем оператор Фурье jw:

$$H^2(w,n) = \frac{w_c^{2n}}{w_c^{2n} + (-1)^n (jw)^{2n}} \tag{26}$$

Замена оператора Фурье jw на оператор Лапласа $p = a + jw$ дает нам передаточную функцию мощности фильтра Баттерворта:

$$H_p(p,n) = \frac{w_c^{2n}}{w_c^{2n} + (-1)^n (p)^{2n}} \tag{27}$$

Введем нормированную комплексную частоту: $S = p/w_c = A + j\Omega$, где $A = a/w_c$, $\Omega = w/w_c$, тогда нормированная передаточная функция мощности примет вид:

$$H_p(S,n) = \frac{1}{1 + (-1)^n (S)^{2n}} \tag{28}$$

Отсюда следует, что на нормированной плоскости S функция $H_p(S,n)$, соответствующая ФНЧ с характеристикой Баттерворта n-го порядка имеет $2n$ полюсов. Полюсы являются корнями уравнения:

$$1 + (-1)^n (S)^{2n} = 0$$

Данные полюса лежат на единичной окружности, рис. 27

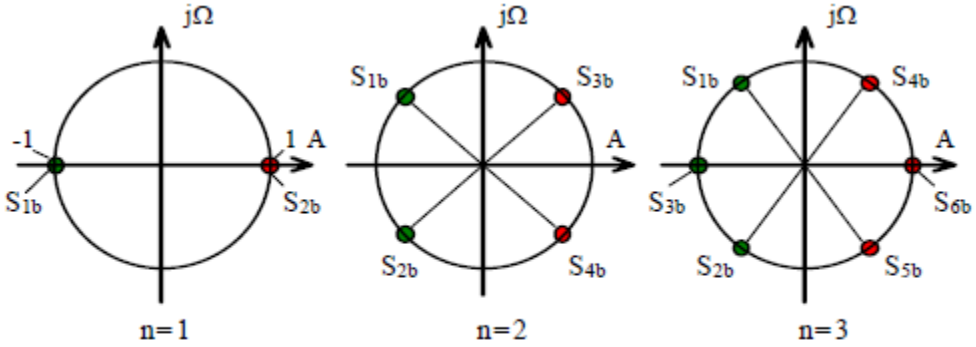


Рис. 27. Полюса фильтра Баттерворта.

В случае n=1 полюсы нормированного коэффициента передачи мощности находят из уравнения: $S^2 = 1$. Его решением является:

$S_1 = -1$ и $S_2 = 1$

В случае n=2 полюсы нормированного коэффициента передачи мощности находят из уравнения: $S^4 = -1$. Его решением является:

$S_1 = \exp(j \cdot 3\pi/4)$; $S_2 = \exp(j \cdot 5\pi/4)$;
$S_3 = \exp(j \cdot \pi/4)$; $S_4 = \exp(j \cdot 7\pi/4)$

Для фильтра третьего порядка из уравнения $S^6 = 1$ получаем шесть полюсов.
Его решением является:

$S_1 = \exp(j \cdot 2\pi/3)$; $S_2 = \exp(j \cdot 4\pi/3)$; $S_3 = -1$;
$S_4 = \exp(j \cdot \pi/3)$; $S_5 = \exp(j \cdot 5\pi/3)$; $S_6 = 1$

Полюсы передаточной функции мощности имеют квадрантную симметрию, а именно их число и конфигурация расположения в обеих полуплоскостях одинаковы. При этом физически реализуемому фильтру с передаточной функцией отвечают только полюсы, расположенные в левой полуплоскости. Их "зеркальные копии" в правой полуплоскости относятся к функции K(-S) и в расчет не принимаются. Таким образом, можно записать нормированную передаточную функцию ФНЧ Баттерворта n-го порядка:

$$H_n(S) = \frac{1}{(S - S_1) \cdot (S - S_2) \cdot \ldots \cdot (S - S_n)} \quad (29)$$

где S_1, S_2, …, S_n – полюсы нормированной передаточной функции мощности $H_p(S, n)$, расположенные на плоскости S в ее левой полуплоскости.

Рассмотрим более подробно передаточную функцию и определить передаточную функцию и амплитудно-частотную характеристику ФНЧ Баттерворта 2-го порядка с частотой среза w_c.

Согласно рисунку рис. 27 при n=2 нормированная передаточная функция должна иметь на плоскости S два полюса в левой полуплоскости:

$S_1 = \exp\left(j \cdot \frac{3\pi}{4}\right) = -0.5\sqrt{2} + 0.5\sqrt{2}$;
$S_2 = \exp\left(j \cdot \frac{5\pi}{4}\right) = -0.5\sqrt{2} - 0.5\sqrt{2}$;

тогда после подстановки в $H_n(S)$ n=2 и упрощения найдем нормированную передаточную функцию:

$$H_n(S) = \frac{1}{S^2 + \sqrt{2}S + 1} \quad (30)$$

Замена переменной S к безразмерной нормированной частоте $\Omega = w/w_c$, дает нормированный коэффициент передачи:

$$H(\Omega) = \frac{1}{-\Omega^2 + j\sqrt{2}\Omega + 1} \quad (31)$$

Соответственно частотный коэффициент передачи для ненормированной частоты будет

$$H(w, w_c) = \frac{w_c^2}{-w^2 + j\sqrt{2}w_c w + 1} \quad (32)$$

Тогда сигнал $U_2(w)$ после фильтра ФНЧ Баттерворта будет определяться следующим образом:

$$U_2(w) = H(w)U_1(w) \quad (33)$$

10.3. Фильтр Баттерворта высоких частот.

Для того чтобы получить передаточную функцию ФВЧ с заданной частотой среза необходимо в передаточной функции нормированного ФНЧ

$$H(\Omega) = \frac{1}{-\Omega^2 + j\sqrt{2}\Omega + 1} \quad (34)$$

Представить безразмерную частоту в виде $\Omega = w_c/w$ (сравните с ФНЧ):

Соответственно частотный коэффициент передачи для ненормированной частоты для фильтра высоких частот будет:

$$H(w, w_c) = \frac{w^2}{-{w_c}^2 + j\sqrt{2}w_c w + 1} \tag{35}$$

Передаточная функция ПФ представляет собой произведение передаточных функций ФНЧ и ФВЧ:

$$H_{\text{пф}}(w, w_c) = H_{\text{фнч}}(w, w_c) H_{\text{фвч}}(w, w_c) \tag{36}$$

Синтез ПФ в данном случае сводится к синтезу ФНЧ и ФВЧ с соответствующими частотами среза и их объединению.

Второй способ заключается в составлении полосового фильтра из звеньев, имеющих частотные характеристики «резонансного» типа. Передаточная функция ПФ с требуемыми граничными частотами $\omega_{п1}$, $\omega_{п2}$, находится из передаточной функции нормированного ФНЧ путем замены оператора s на новый оператор $(s_2+w_0^2)/(Bs)$, где B= $\omega_{п1}$, $\omega_{п2}$ –ширина полосы пропускания; w_0 – средняя резонансная частота.

$$H(s) = H_{\text{н}}\left(\frac{s^2 + w_0^2}{Bs}\right)$$
$$B = w_{\text{п2}} - w_{\text{п1}};\ w_0 = \sqrt{w_{\text{п2}} w_{\text{п1}}} \tag{37}$$

Построение заграждающих фильтров может осуществляться двумя способами. Первый заключается в составлении заграждающего фильтра из ФНЧ и ФВЧ, включенных параллельно. Передаточная функция такого заграждающего фильтра представляет собой сумму передаточных функций ФНЧ и ФВЧ:

$$H_{\text{зф}}(S) = H_{\text{фнч}}(s) + H_{\text{фвч}}(s) \tag{38}$$

Синтез ЗФ в данном случае сводится к синтезу ФНЧ и ФВЧ, граничные частоты которых рассчитываются по заданным граничным частотам полосы задерживания. Второй способ заключается в составлении заграждающего фильтра из звеньев, имеющих частотные характеристики заграждающего типа. Синтез заграждающего фильтра данного типа осуществляется с использованием нормированного ФНЧ. Передаточная функция заграждающего фильтра с требуемыми граничными частотами задерживания $\omega_{з1}$, $\omega_{з2}$ может быть получена из передаточной функции нормированного ФНЧ путем замены оператора s на новый оператор $Bs/(s_2+w_0^2)$, то есть

$$H(s) = H_{\text{н}}\left(\frac{Bs}{s^2 + w_0^2}\right)$$
$$B = w_{\text{з2}} - w_{\text{з1}};\ w_0 = \sqrt{w_{\text{з2}} w_{\text{з1}}} \tag{39}$$

10.4. Задания

На вход передатчика подается сигнал косинуса с частотами 50, 150 и 450 Гц. Определите спектр сигнала с помощью стандартной функции быстрого преобразования Фурье.

(а) построить фильтр Баттерворта нижних частот (ФНЧ) к сигналу, убедиться, что происходит подавление высоких частот сигнала, воспользовавшись формулой (32).

(б) построить фильтр высоких частот (ФВЧ) к сигналу, убедиться, что происходит подавление низких частот сигнала, воспользовавшись формулой (35).

(в) построить полосовой фильтр (ПФ), воспользовавшись формулой (36) и (37), настроенный для пропускания частоты 150 Гц.

МЕТОДЫ РЕШЕНИЯ ИНТЕГРАЛЬНЫХ УРАВНЕНИЙ

Линейное уравнение Фредгольма II рода имеет следующий вид:

$$y(x) - \lambda \int_a^b K(x,s)y(s)\,ds = f(x) \tag{40}$$

Здесь y(x) – неизвестная функция, K(x, s) – ядро интегрального уравнения, $f(x)$ – свободный член (правая часть) интегрального уравнения. Для удобства анализа в интегральном уравнении (40) принято выделять числовой параметр λ, который называют параметром интегрального уравнения.

11.1. Метод квадратур.

Найдем приближенное решение уравнения (40) методом квадратур. Построим на отрезке $[a, b]$ сетку с узлами $x_1, x_2, \dots, x_n$. Запишем уравнение (40) в узлах сетки:

$$y(x_i) - \lambda \int_a^b K(x_i,s)y(s)\,ds = f(x_i), \qquad i = 1,2,\dots n. \tag{41}$$

Аппроксимируем интегралы в равенствах (41) конечными суммами с помощью одной из квадратурной формул:

$$y_i - \lambda \sum_{j=1}^{n} A_j K_{ij} y_j = f_i, \qquad i = 1,2,\dots n. \tag{42}$$

Здесь $y_i = \hat{y}(x_i)$, $f_i = f(x_i)$, $K_{ij} = K(x_i, x_j)$, $\hat{y}$ = приближенное решение к искомой функции y, и A_j - веса квадратурной формулы.Решение системы уравнения (42) дает приближенное значение искомой функции в узлах x_i. По ним с помощью интерполяции можно построить приближенное решение интегрального уравнения (40) на всем отрезке $[a, b]$.

Пусть $\lambda = 1$, а сетка $x_1, x_2, \dots, x_n$ равномерная с шагом h. Используем квадратурную формулу трапеций. Тогда система линейных алгебраических уравнений (42) примет следующий вид:

$$y_i - h \sum_{j=1}^{n} w_j K_{ij} y_j = f_i, \qquad i = 1,2,\dots n. \tag{43}$$

где $w_1 = w_n = 1/2, w_j = 1$ при $j = 2,3,\dots n-1$.

Рассмотрим пример решения упражнения из книги [16].

Дано уравнение (40) с границами интегрирования $a = -\pi$, $b = \pi$, параметром $\lambda = 3/(10\pi)$ и ядром, представленной формулой (44):

$$K(x,s) = \frac{1}{0.64cos^2\left(\frac{x+s}{2}\right) - 1} \tag{44}$$

И правой частью $f(x) = 25 - 16sin^2(x)$. Точное решение уравнения (40) будет являться функция $y(x) = 17/2 - (128/17)cos(2x)$. Найти приближенное решение этого уравнения методом квадратур, основанным на использовании формулы трапеций с равномерной сеткой и шагом $h = \pi/30$. Сравнить решение с точным решением.

```
import  numpy as np
from scipy.integrate  import odeint
import matplotlib.pyplot as plt
import math

h=math.pi/30
a=-math.pi
```

```
b =math.pi+0.001
lam = 3/(10*math.pi)

x=np.arange (a, b, h)
x=x.reshape(len(x),1)
n=len(x)

K = lambda x1,s: 1/(0.64*(np.cos((x1+s)/2))**2-1)*lam
f = lambda x1:25-16*(np.sin(x1))**2
y_exact=lambda x1: 17/2+128/17*np.cos(2*x1)

y=[] # точное решение
for i in range(n):
   y.append([])  # создаем пустую строку
   y[i].append(y_exact(x[i]))
y=np.array(y).reshape(n,1) # точное решение

def Fred_2(K,f,a,b,h):
   x=np.arange (a, b, h)
   x=x.reshape(len(x),1)
   n=len(x)
   wt=1/2
   wj=1
   A=np.zeros((n, n))

   for i in range(n):
      A[i][0]=-h*wt*K(x[i],x[0])
      for j in range(1,n-1,1):
         A[i][j]= -h*wj*K(x[i],x[j])
      A[i][n-1]= -h*wt*K(x[i],x[n-1])
      A[i][i]= A[i][i]+ 1

   B = np.zeros((n,1))
   for j in range(n):
      B[j][0] = f(x[j])

   y=np.linalg.solve(A, B)
   return y

y_approx=Fred_2(K,f,a,b,h)

plt.plot(x,y, '-g',linewidth=2, label='y_exact') # график точного решения
plt.plot(x,y_approx, 'or',  label='y_approx') # график найденного решения
plt.xlabel("x")
plt.ylabel("y")
plt.legend(bbox_to_anchor=(1, 1), loc='best')
plt.ylim([0,max(y)+2])
plt.show()
```

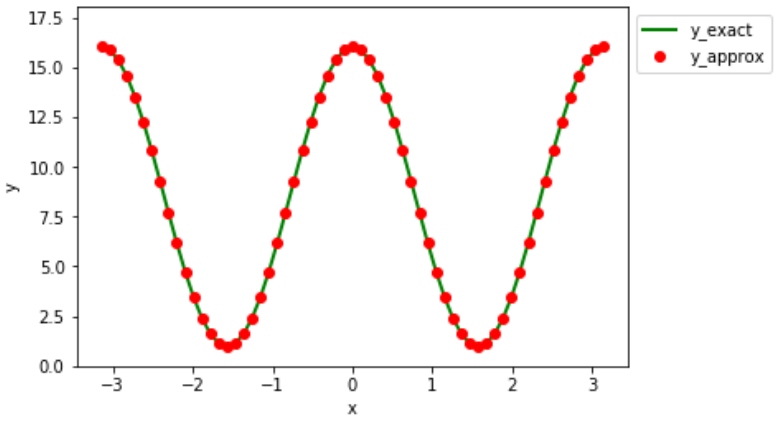


Рис. 28. Сплошной линией обозначено точное решение, круглыми символами – приближенное решение.

11.2. Метод вырожденных ядер.

Ядро интегрального уравнения (40) называют вырожденным, если оно имеет следующий вид:

$$K(x,s) = \sum_{i=1}^{m} \alpha_i(x)\beta_i(s) \tag{45}$$

Будем считать, что $\{\alpha(x)\}_{i=1}^{m}$, $\{\beta(x)\}_{i=1}^{m}$ – линейно независимые системы непрерывных функций на отрезке $[a,b]$.

Тогда получим следующее представление решение уравнения с вырожденным ядром:

$$y(x) = f(x) + \lambda \sum_{i=1}^{m} c_i \alpha_i(x) \tag{46}$$

где неизвестные коэффициенты c_i могут быть определены из формул:

$$\mathrm{c}_i = \int_a^b \beta_i(s) y(s)\, ds, \qquad i = 1,2,\dots m. \tag{47}$$

Система функций $\{\alpha(x)\}_{i=1}^{m}$ линейно независимая, следовательно все коэффициенты в этой комбинации равны нулю:

$$c_i - \lambda \sum_{j=1}^{m} c_j \int_a^b \alpha_i(s)\beta_i(s) ds = \int_a^b \beta_i(s) f(s) ds \qquad i = 1,2,\dots m. \tag{48}$$

Обозначим:

$$a_{ij} = \int_a^b \alpha_i(s)\beta_i(s) ds\,, \quad i,j = 1,2,\dots m. \tag{49}$$

и функции:

$$f_i = \int_a^b \beta_i(s) f(s) ds\,, \quad i = 1,2,\dots m. \tag{50}$$

Запишем равенства (48) в виде системы линейных алгебраических уравнений относительно неизвестных c_i.

$$\mathrm{c}_i - \lambda \sum_{j=1}^{m} a_{ij} \mathrm{c}_j = f_i, \quad i = 1,2,\dots m. \tag{51}$$

или в матричном виде:

$$\begin{pmatrix} 1-\lambda a_{11} & -\lambda a_{12} & \vdots & -\lambda a_{1m} \\ -\lambda a_{21} & 1-\lambda a_{22} & \vdots & -\lambda a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ -\lambda a_{m1} & -\lambda a_{m2} & \vdots & 1-\lambda a_{mm} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} \quad (52)$$

Если число λ регулярное, то определитель матрицы этой системы отличен от нуля, и она имеет единственное решение.

Таким образом, алгоритм решения интегрального уравнения (40) с вырожденным ядром (45) состоит в вычислении интегралов (49) и (50) и решении системы линейных алгебраических уравнений (52). Решение уравнения (40) записывается аналитически в виде уравнения (46).

В качестве примера решим уравнение с вырожденным ядром:

$$y(x) - \int_0^1 (1 + 2xs)y(s)\,ds = -\frac{1}{6}x - \frac{1}{2} \quad x \in [0,1]$$

Точное решение данного уравнения будет функция, см [16]:

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

Рассмотрим пример решения на языке Python:

```
import  numpy as np
from scipy.integrate  import odeint
import matplotlib.pyplot as plt
import math
import scipy.integrate as integrate
import scipy.special as special

a = 0
b = 1.001
h=0.05
Lambda = 1
x=np.arange (a, b, h)
x=x.reshape(len(x),1)
n=len(x)

alpha = lambda t: [1,2*t]
beta= lambda t: [1, t]
f =lambda t: -t/6 - 1/2
y_exact=lambda t: t+0.5 # точное решение

y=[] # точное решение
for i in range(n):
   y.append([])  # создаем пустую строку
   y[i].append(y_exact(x[i]))
y=np.array(y).reshape(n,1) # точное решение

def bfun(t,m,f):
      return beta(t)[m]*f(t)

def Aijfun(t,m,k):
      return beta(t)[m]*alpha(t)[k]
```

```
def Solve(f,t,Lambda):
    m=len(alpha(0)) # определяем размер alpha
    M=np.zeros((m,m))
    r=np.zeros((m,1))

    for i in range(m):
        r[i]=integrate.quad(bfun, a, b,args=(i,f))[0]
        for j in range(m):
            M[i][j]=-Lambda*integrate.quad(Aijfun, a, b,args=(i,j))[0]
    for i in range(m):
        M[i][i] =M[i][i]+1

    c=np.linalg.solve(M, r)

    return Lambda*(c[0]*alpha(t)[0]+c[1]*alpha(t)[1])+f(t)

y_approx=Solve(f,x,Lambda)

plt.plot(x,y, '-g',linewidth=2, label='y_exact') # график точного решения
plt.plot(x,y_approx, 'or',  label='y_approx') # график найденного решения
plt.xlabel("x")
plt.ylabel("y")
plt.legend('1',fontsize=12)
plt.legend(bbox_to_anchor=(1, 1), loc='best')
plt.ylim([0,max(y)+0.1])
plt.show()
```

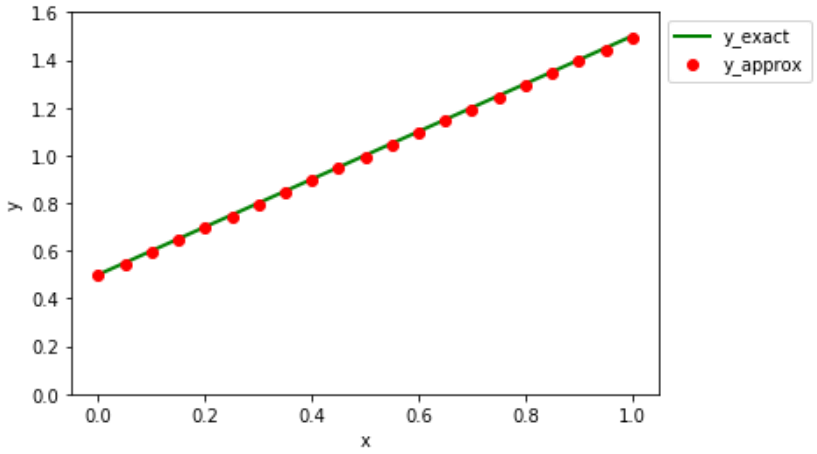


Рис. 29. Сплошной линией обозначено точное решение, круглыми символами – приближенное решение.

11.3. Задания.

1. С помощью метода квадратур найдите приближенное решение уравнения

$$y(x) = \frac{5}{6}x + \frac{1}{2}\int_0^1 xsy(s)ds \quad x \in [0,1]$$

Если его точное решение $y(x) = x$, см см [16].

2. Решить следующие уравнение с помощью аппроксимации ядра вырожденным:

$$y(x) + \int_0^1 x(e^{xs} - 1)y(s)ds = e^x - 1; \quad x \in [0,1]$$

Замечание: если ядро интегрального уравнения невырожденное, но достаточно гладкое, тогда его можно аппроксимировать вырожденным, разложив его в ряд Тейлора.

3. **Метод Галеркина-Петрова.** Выражение для невязки уравнения Фредгольма II рода имеет следующий вид:

$$Ry(x) = y(x) - \lambda \int_a^b K(x,s)y(s)\,ds - f(x), \qquad x \in [a,b]. \tag{53}$$

Если $y(x)$ – точное решение уравнения (40), тогда невязка будет равна нулю.

Пусть решение уравнения (40) рассматривается в линейном пространстве со скалярным произведением. Выберем в нем две системы линейно-независимых функций $\varphi_i(x)$ и $\psi_i(x)$, $i = 1,2,\dots n$. Приближенное решение уравнения запишем в виде:

$$\hat{y}(x) = f(x) + \sum_{j=1}^{n} c_j \varphi_j(x) \tag{54}$$

где c_j – подлежащие определению коэффициенты. Ясно, что если невязка для функции $\hat{y}(x)$ равна нулю, т.е.$\hat{y}(x)$ является точным решением, то она ортогональна каждой из функций $\psi_i(x)$, Используем эти же условия ортогональности для обеспечения малости невязки уравнения с приближенным решением $\hat{y}(x)$:

$$(R\hat{y}, \psi_i) = 0,, \quad i = 1,2,\dots n \tag{55}$$

Пусть скалярное произведение определено формулой:

$$(f,g) = \int_a^b f(x)g(x)\,dx \tag{56}$$

Тогда равенство (55) представляет собой систему линейных алгебраических уравнения относительно неизвестных коэффициентов c_j:

$$\sum_{j=1}^{n} a_{ij}c_j = b_i,, \quad i = 1,2,\dots n \tag{57}$$

Здесь

$$a_{ij} = \int_a^b \varphi_i(x)\psi_i(x)dx - \lambda \int_a^b \psi_i(x) \int_a^b K(x,s)\varphi_j(s)\,dsdx \tag{58}$$

и

$$b_i = \lambda \int_a^b \psi_i(x) \int_a^b K(x,s)f(s)\,dsdx \tag{59}$$

Решив систему уравнений (57) найдем коэффициенты в представлении приближенного решения (54). В наиболее общем виде этот метод обоснован, например, в [17].

(а) Решить с помощью метода Галеркина-Петрова уравнение:

$$y(x) = 1 + \int_{-1}^{1} (x^2 + xs)y(s)ds\,; \quad x \in [-1,1]$$

Точным решением этого уравнения является функция: $y(x) = 1 + 6x^2$ (см, [16]). Приближенное решение искать в виде:

$$\tilde{y}(x) = 1 + c_1\varphi_1(x) + c_2\varphi_2(x)$$

где $\varphi_1(x) = x$, $\varphi_2(x) = x^2$, из условия ортогональности невязки функциям $\psi_1(x) = 1$ и $\psi_2(x) = x$. Напишите программу, которая найдет решение методом Галеркина-Петрова, и сопоставьте его с точным решением. Точное решение $y(x) = 1 + 6x^2$ задачи и искомое решение задачи представлено на рис. 30.

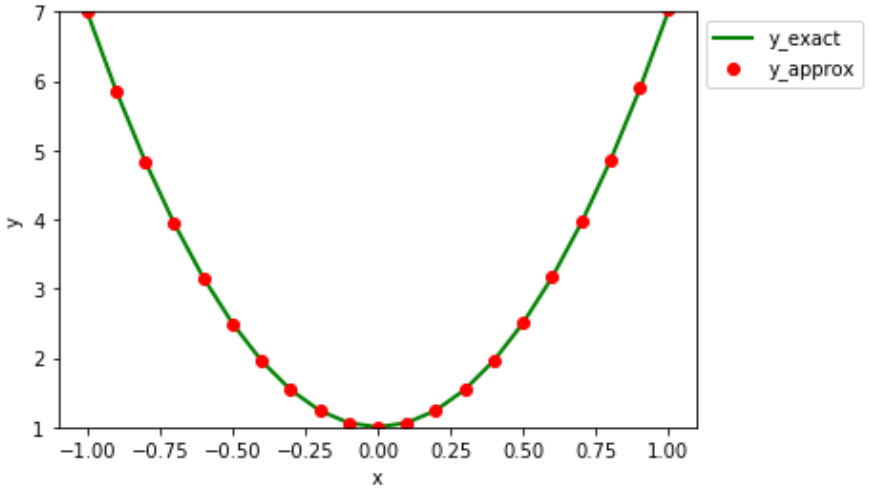


Рис. 30. Сплошной линией обозначено точное решение, круглыми символами – приближенное решение.

(б) Частным случаем метода Галеркина-Петрова является метод Бубнова-Галеркина. В этом методе полагают $\varphi_i(x) = \psi_i(x)$ для всех $i = 1,2,\dots,n$. Напишите компьютерную программу для решения задачи (а). Приближенное решение искать в виде:

$$\tilde{y}(x) = 1 + c_1\varphi_1(x) + c_2\varphi_2(x)$$

где $\varphi_1(x) = x$, $\varphi_2(x) = x^2$.

ТЕСТОВЫЕ ЗАДАНИЯ

В данном разделе представлены тестовые задания, которые могут быть использованы для проверки знаний по основам программирования на языке Python.

12.1. Тест 1.

1. Вычислить следующие выражения:(а) $\sqrt[3]{1331}$; (б) $\tan(\pi/3)$; (в) $exp(\sin(2))$; (г) $i + 1/i$.
2. Построить график функции

$$f(x) = \frac{\cos(x)}{2 + \sin(x)}, \text{если } -2\pi \le x \le 2\pi$$

Постройте график красной линией, подпишите оси координат.

3. Выведите на экран список 7, 11, 15, 19, 23, 27, 31. (а) записав список по определению; (б) используя функцию range(); (г) с использованием цикла for.
4. Напишите функцию, которая вычисляет сумму $(x_1 + y_1, x_2 + y_2, \dots x_n + y_n)$ для массивов $x = (x_1, x_2, \dots, x_n)$ и $y = (y_1, y_2, \dots, y_n)$, используя цикл for. Проверить результат для массивов (3,4,5) и (-1,5,4).
5. Постройте график, заданной кусочным образом:

$$y(x) = \begin{cases} \pi \sin(x), & -2\pi \le x \le -\pi \\ \pi - \text{x}, & -\pi \le x \le \pi \\ \pi sin^3 x, & \pi \le x \le 2\pi \end{cases}$$

12.2. Тест 2.

1. Вычислить следующие выражения:(а) $\sqrt[4]{2401}$; (б) $sin(\pi/3)$; (в) $exp(3/4))$; (г) $\cos(2 + i)$.
2. Построить график функции

$$f(x) = x^2 \cos(3x), \text{если } 0 \le x \le 20$$

Постройте график зеленой линией, подпишите оси координат.

3. Выведите на экран список 1, 3, 5, 7, 9, 11 (а) записав список по определению; (б) используя функцию range(); (г) с использованием цикла for.
4. Напишите функцию, которая вычисляет евклидово расстояние n-мерного вектора x=(x_1, x_2,…, x_n) от начала координат по формуле $dist = \sum_{k=1}^{n} x_k^2$ На входе вашей функции должен быть список, представляющий вектор x. Используйте вашу функцию для вычисления евклидовых расстояний 3-мерных векторов (0,3, −4) и (6, −1,3) путем запуска следующих команд:

print(dist([0,3,-4]))
print(dist([6,-1,3]))

5. Постройте график, заданной кусочным образом:

$$g(x) = \begin{cases} x & \text{для } 2 \le x \le 4 \\ 0 & \text{иначе} \end{cases}$$

Контрольные вопросы

Тематика вопросов на устном дифференциальном зачете соответствует избранным разделам (темам) дисциплины «Цифровая обработка сигналов»:

- Язык Python. Типы данных, операции, операторы. Встроенные типы данных и классификация коллекий. Особые методы сравнения множеств (set, frozenset).
- Библиотека numpy для реализации математических объектов и вычислений.
- Конвертация одного типа коллекции в другой. Индексированные коллекции.
- Синтаксис и особенность среза. Сортировка элементов коллекции. Добавление и удаление элементов изменяемых коллекций.
- Генераторы выражений. Определения и классификация. Вложенные циклы и генераторы.
- Основные стандартные модули Python.
- Операторы условия и сравнения в Python.
- Решение дифференциальных уравнений в Python.
- Численное интегрирование в Python.
- Дискретизация аналоговых сигналов.
- Тригонометрический ряд Фурье. Комплексный ряд Фурье. Явление Гиббса.
- Основные свойства преобразования Фурье. Теорема Парсеваля.
- Дискретное преобразование Фурье. Быстрое преобразование Фурье.
- Фурье преобразование в Python.
- Фильтрация сигналов. Фильтр Баттерворта. Фильтр нижних частот (ФНЧ). Фильтр высоких частот, полосовой и заграждающий фильтр.
- Задача фильтрации. Базисные фильтры. Задача аппроксимации. Типовые ФНЧ.
- Фильтр Баттерворта. Нули и полюса нормированного ФНЧ Баттерворта.
- Фильтр Чебышева первого рода
- Денормирование и трансформация цифровых фильтров.
- Методы решения уравнений Фредгольма II рода.
- Метод квадратур.
- Метод вырожденных ядер.

Заключение

Данное методическое пособие разработано для курса «Цифровая обработка сигналов». Задачей данного методического пособия является изучение методов анализа и обработки сигналов, способов их преобразования и передачи по каналам связи, освоение теоретических основ математического аппарата цифровой обработки одно- и многомерных сигналов, освоение современных программных инструментов. Ставится задача сформировать навыки осмысления результатов физического эксперимента, построения математических моделей исследуемых процессов, познакомить студентов с алгоритмами обработки сигналов, основами цифровой фильтрации и принципов работы ЦАП и АЦП.

Данный практический курс поможет читателю лучше понять основы цифровой обработки сигналов на языке Python. Кроме того, в этот материал входит перевод различных научных статей, компиляция информации из достоверных источников и литературы по тематике цифровой обработки сигналов, а также официальная документация по прикладным пакетам и встроенным функциям библиотек scipy и numpy языка Python.

Литература

1. Электронный ресурс: https://docs.python.org/2/faq/general.html#why-is-it-called-python
2. Peters T. Pep 20–the zen of python //Python Enhancement Proposal. – 2004.
3. Электронный ресурс: https://habr.com/ru/post/320288/
4. Электронный ресурс (на английском): https://wiki.python.org/moin/HowTo/Sorting#Key_Functions
5. Электронный ресурс (на английском): https://wiki.python.org/moin/HowTo/Sorting#Sort_Stability_and_Complex_Sorts
6. Электронный ресурс: https://ru.wikipedia.org/wiki/Pandas
7. Электронный ресурс: https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html
8. Winkler P. Mathematical puzzles: a connoisseur's collection. – AK Peters/CRC Press, 2003.
9. Стюарт И. Величайшие математические задачи. – Альпина Паблишер, 2015.
10. Электронный ресурс: https://en.wikipedia.org/wiki/Lambda_calculus
11. Электронный ресурс: https://ru.wikipedia.org/wiki/Теорема_о_промежуточном_значении
12 .Доля П. Г. Введение в научный Python //Харьковский национальный университет. Факультет математики и информатики. – 2016. – С. 265.
13. Cooley J. W., Tukey J. W. An algorithm for the machine calculation of complex Fourier series //Mathematics of computation. – 1965. – Т. 19. – №. 90. – С. 297-301.
14. Электронный ресурс: http://jakevdp.github.io/blog/2012/09/05/quantum-python/
15. Электронный ресурс: https://ru.wikipedia.org/wiki/Белый_шум
16. Верлань А. Ф., Сизиков В. С. Интегральные уравнения: методы алгоритмы программы: Справочное пособие. – Наук. думка, 1986.
17 Красносельский М. А. и др. Приближенное решение операторных уравнений. – 1969.