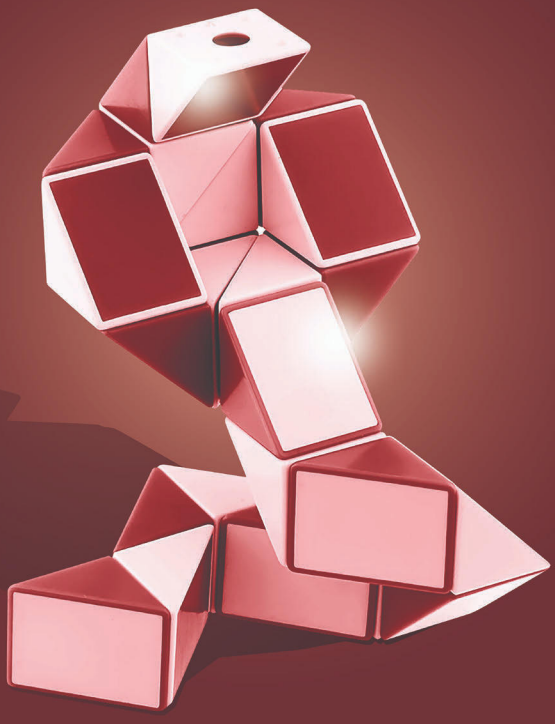
М. В. Сысоева, И. В. Сысоев

библиотека alt



Программирование для «нормальных» с нуля на языке Python Часть 2

Учебник по программированию

В серии: Библиотека ALT

М. В. Сысоева, И. В. Сысоев

Программирование для «нормальных» с нуля на языке Python

В двух частях

Часть 2

Учебник

Москва
Базальт СПО
МАКС Пресс
2023

УДК 004.43(075.8)
ББК 22.1я73
С95

https://elibrary.ru/gzazeu

Рецензенты:

В. И. Пономаренко – доктор физико-математических наук, профессор, ведущий научный сотрудник Саратовского филиала Института радиотехники и электроники имени В. А. Котельникова РАН;

В. В. Матросов – доктор физико-математических наук, профессор, декан радиофизического факультета Национального исследовательского Нижегородского государственного университета им. Н. И. Лобачевского;

Г. В. Курячий – преподаватель факультета ВМК МГУ имени М. В. Ломоносова, автор курсов по Python для ВУЗов и Вечерней математической школы, разработчик компании «Базальт СПО»

Сысоева М. В., Сысоев И. В.

С95 **Программирование для «нормальных» с нуля на языке Python** : Учебник. В двух частях. Часть 2 / М. В. Сысоева, И.В. Сысоев ; отв. ред. В. Л. Черный : – Москва : Базальт СПО; МАКС Пресс, 2023. – 184 с.: ил. – (Библиотека ALT).
ISBN 978-5-317-06945-2
ISBN 978-5-317-06947-6 (Часть 2)

Книга – учебник, задачник и самоучитель по алгоритмизации и программированию на языке Python. Она не требует предварительных знаний в области программирования и может использоваться для обучения «с нуля».

Издание адресовано студентам, аспирантам и преподавателям инженерных и естественно-научных специальностей вузов, школьникам старших классов и учителям информатики. Обучение языку в значительной степени строится на примерах решения задач обработки результатов радиофизического и биологического экспериментов.

Ключевые слова: программирование; численные методы; алгоритмы; графики; Python; numpy.

УДК 004.43(075.8)
ББК 22.1я73

Сайт книги: `http://www.altlinux.org/Books:Python-sysoeva-ed2`

ISBN 978-5-317-06947-6 (Часть 2)
ISBN 978-5-317-06945-2

Оглавление

Предисловие

Почему стоит продолжить изучение Python? Python часто называют скриптовым языком, языком прототипирования и пригодным для разработки «на скорую руку». Много раз мы слышали о том, что его вообще нельзя давать новичкам, потому что он поставит им неправильное мышление или потому что всё равно потом придётся учить что-то «настоящее» и «промышленно востребованное», как правило, статически типизируемое и компилируемое, что «не тормозит». Это — распространённая точка зрения в профессиональном сообществе программистов (хотя и не единственная), то есть среди людей, для которых написанная программа является конечным продуктом их труда. Поскольку мы предполагаем, что основная наша аудитория — это люди, программированию не чуждые, но зарабатывающие на жизнь другим, будь то инженеры самых разных мастей, учёные от физиков до лингвистов, работники статистических служб, аналитики банков и инвестиционных компаний и прочие, мы смело отбросим данное суждение и постараемся показать, что в Python есть практически всё, что вам нужно для вашей работы, часто с вариантами.

В этой второй части мы рассмотрим:

1. создание собственных функций и модулей,
2. функциональное и многопоточное программирование,
3. интерфейс к базам данных,
4. графический интерфейс пользователя и принципы объектно-ориентированного программирования на его примере,
5. исследование динамических систем различными средствами `scipy`,
6. параллельное и многопоточное программирование.

Сведения об авторах

- Сысоева Марина Вячеславовна — кандидат физико-математических наук, доцент кафедры «Радиоэлектроника и телекоммуникации» Саратовского государственного технического университета имени Гагарина Ю.А.

- Сысоев Илья Вячеславович — доктор физико-математических наук, профессор кафедры системного анализа и автоматического управления Саратовского национального исследовательского государственного университета имени Н.Г. Чернышевского.

Сведения о рецензентах

- Пономаренко Владимир Иванович — доктор физико-математических наук, профессор, ведущий научный сотрудник Саратовского филиала Института радиотехники и электроники имени В. А. Котельникова РАН.
- Матросов Валерий Владимирович — доктор физико-математических наук, профессор, декан радиофизического факультета Национального исследовательского Нижегородского государственного университета им. Н. И. Лобачевского.
- Курячий Георгий Владимирович — преподаватель факультета ВМК Московского Государственного Университета им. М. В. Ломоносова, автор курсов по Python для ВУЗов и Вечерней математической Школы, разработчик компании «Базальт СПО».

Глава 8

Функции

8.1 Функции в программировании

Функции в программировании можно представить как изолированный блок кода, обращение к которому в процессе выполнения программы может быть многократным. Зачем нужны такие блоки инструкций? В первую очередь, чтобы сократить объем исходного кода: рационально вынести часто повторяющиеся выражения в отдельный блок и затем по мере надобности обращаться к нему.

Для того, чтобы в полной мере осознать необходимость использования функций, приведём сложный, но чрезвычайно полезный пример вычисления корня нелинейного уравнению с помощью метода деления отрезка пополам.

Пусть на интервале $[a; b]$ имеется ровно 1 корень уравнения $f(x) = 0$. Значит, $f(a)$ и $f(b)$ имеют разные знаки. Используем этот факт. Найдём $f(c)$, где $c = \frac{a+b}{2}$. Если $f(c)$ того же знака, что и $f(a)$, значит корень расположен между c и b, иначе — между a и c.

Пусть теперь начало нового интервала (будь то a или c в зависимости от того, где находится корень) обозначается a_1 (что означает после первой итерации), а конец, соответственно, b_1. Исходные начало и конец также будем обозначать a_0 и b_0 для общности. В результате нам удасться снизить неопределённость того, где находится корень, в два раза.

Такой процесс можно повторять сколько угодно раз (при расчёте на компьютере столько, сколько позволяет точность представления данных ЭВМ), последовательно заужая интервал, в котором находится корень. Обычно процесс заканчивают по достижении на n-ой итерации интервалом $[a_n; b_n]$ величины менее некоторого заранее заданного ε.

Итак, напишем программу для вычисления корня нелинейного уравнения $f(x) = x^2 - 4$:

```
from math import *
a = -10
b = 10
```

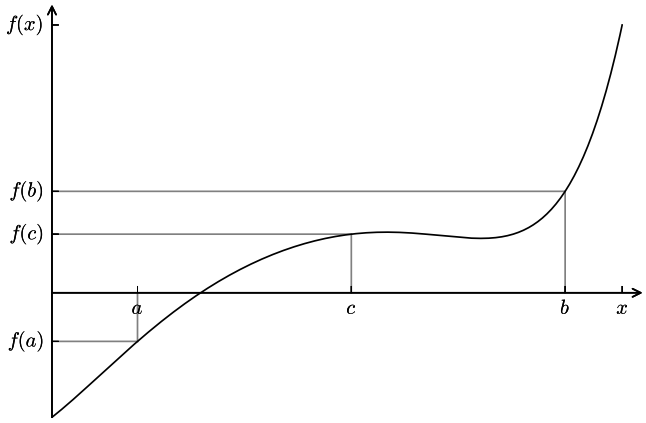


Рис. 8.1. Иллюстрация к методу деления отрезка пополам.

```
while (b-a) > 10**(-10):
    c = (a+b)/2
    f_a = a**2-4
    f_b = b**2-4
    f_c = c**2-4
    if f_a*f_c > 0:
        a = c
    else:
        b = c
print((a+b)/2)
```

Если мы захотим вычислить корень другой нелинейной функции, например, $f(x) = x^2 + 4x + 4$, то придётся переделывать выражения сразу в трёх строчках. Кажется логичным выделить наше нелинейное уравнение в отдельный блок программы. Перепишем программу с помощью функции:

```
from math import *
def funkcija(x):
    f = x**2+4*x+4
    return f
a = -10
b = 10
while (b-a) > 10**(-10):
    c = (a+b)/2
    f_a = funkcija(a)
```

```
    f_b = funkcija(b)
    f_c = funkcija(c)
    if f_a*f_c > 0:
        a = c
    else:
        b = c
print((a+b)/2)
```

Программный код подпрограммы описывается единожды перед телом основной программы, затем из основной программы можно им пользоваться многократно. Обращение к этому программному коду из тела основной программы осуществляется по его имени (имени подпрограммы).

Инструкция `def` — это команда языка программирования Python, позволяющая создавать функцию; `funkcija` — это имя функции, которое (так же как и имена переменных) может быть почти любым, но желательно осмысленным. После в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми. Далее идет двоеточие, обозначающее окончание заголовка функции (аналогично с условиями и циклами). После заголовка с новой строки и с отступом следуют выражения тела функции. В конце тела функции обычно присутствует инструкция `return`, после которой идёт значение или выражение, являющееся результатом работы функции. Именно оно будет подставлено в главной (вызывающей) программе на место функции. Принято говорить, что функция «возвращает значение», в данном случае — результат вычисления выражения $x^2 + 4x + 4$ в конкретной точке x. В других языках программирования такая инструкция называется также **функцией**, а инструкция, которая ничего не возвращает, а только производит какие-то действия, называется **процедурой**[1], например:

```
def procedura(x):
    f = x**2+4*x+4
    print(f)
```

Те же самые инструкции можно переписать в виде функции `Bisection`, которой передаются данные из основной программы, и которая возвращается корень уравнения с заданной точностью:

```
from math import *
def function(x):
    f = x**2-4
    return f
def Bisection(a, b, e):
```

[1]На самом деле никакого разделения на функции и процедуры в Python нет. Просто те функции, в которых возвращаемое значение не указано, возвращают специальное значение `None`. Более того, даже если функция какой-то результат возвращает, вы можете его проигнорировать и использовать её как процедуру (в этом случае она, конечно, должна делать что-то полезное кроме вычисления возвращаемого значения).

```
    while (b-a) > e:
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
    return ((a+b)/2)
A = -10
B = 10
E = 10**(-15)
print(Bisection(A, B, E))
```

Вывод программы:

```
-2.0000000000000004
```

В Python результатом функции может быть только одно значение. Если необходимо в качестве результата выдать значения сразу нескольких переменных, используют кортеж. Продемонстрируем это, дополнив программу вычислением количества шагов, за которые достигается значение корня с заданной точностью:

```
from math import *
def function(x):
    f = x**2-4
    return f
def Bisection(a, b, e):
    n = 0
    while (b-a) > e:
        n = n + 1
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
    return ((a+b)/2, n)
A = -10
B = 10
E = 10**(-15)
print(Bisection(A, B, E))
```

В качестве результата выдаётся кортеж из двух чисел: значения корня и количества шагов, за которое был найден этот корень:

```
(-2.0000000000000004, 55)
```

Выражения тела функции выполняются лишь тогда, когда она вызывается в основной ветке программы. Так, например, если функция присутствует в исходном коде, но нигде не вызывается, то содержащиеся в ней инструкции не будут выполнены ни разу.

8.2 Параметры и аргументы функций

Часто функция используется для обработки данных, полученных из внешней для нее среды (из основной ветки программы). Данные передаются функции при её вызове в скобках и называются аргументами. Однако чтобы функция могла «взять» передаваемые ей данные, необходимо при её создании описать параметры (в скобках после имени функции), представляющие собой переменные.

Пример:

```
def summa(a,b):
    c = a + b
    return c
num1 = int(input('Введите первое число: '))
num2 = int(input('Введите второе число: '))
summa(num1, num2)
```

Здесь `num1` и `num2` суть *аргументы* функции, `a` и `b` суть *параметры* функции.

В качестве аргументов могут выступать как числовые или строковые константы вроде `12.3`, `-9` или `'Hello'`, так и переменные или выражения, например, `a+2*i`.

8.2.1 Обязательные и необязательные аргументы

Аргументы функции могут быть обязательными или необязательными. Для всех необязательных аргументов необходимо указать значение по умолчанию.

Рассмотрим функцию, возводящую один свой параметр в степень, заданную другим:

```
def Degree(x, a):
    f = x**a
    return f
```

Если мы попробуем вызвать эту функцию с одним аргументом вместо двух, получим ошибку:

```
>> Degree(3)
Traceback (most recent call last):
```

```
  File "<pyshell#4>", line 1, in <module>
    Degree(3)
TypeError: Degree() missing 1 required positional argument: 'a'
```

Интерпретатор недоволен тем, что параметру с именем `a` не сопоставили ни одного значения. Если мы хотим, чтобы по умолчанию функция возводила `x` в квадрат, можно переопределить её следующим образом:

```
def Degree(x, a=2):
    f = x**a
    return f
```

Тогда наш вызов с одним аргументом станет корректным:

```
>>> Degree(3)
9
```

При этом мы не теряем возможность использовать функцию `Degree` для возведения в произвольную степень:

```
>>> Degree(3, 4)
81
```

В принципе, все параметры функции могут иметь значение по умолчанию, хотя часто это лишено смысла. Параметры, значение по умолчанию для которых не задано, называются *обязательными*. Важно помнить, что обязательный параметр не может стоять после параметра, имеющего значение по умолчанию. Попытка написать функцию, не удовлетворяющую этому требованию, приведёт к синтаксической ошибке:

```
>>> def Degree(x=2, a):
    f = x**a
    return f
SyntaxError: non-default argument follows default argument
```

8.2.2 Именованные аргументы

Бывает, что у функции много или очень много параметров. Или вы забыли порядок, в котором они расположены, но помните их смысл. Тогда можно обратиться к функции, используя имена параметров как ключи. Пусть у нас есть следующая функция:

```
def Clothing (Dress, ColorDress, Shoes, ColorShoes):
    S = 'Сегодня я надену ' + ColorDress + '␣'  \
      + Dress + '␣и '+ ColorShoes + '␣' + Shoes
    return S
```

Теперь вызовем нашу функцию, с аргументами не по порядку:

```
print(Clothing(ColorDress='красное', Dress='платье',
               ColorShoes='чёрные', Shoes='туфли'))
```

Будет выведено:

```
Сегодня я надену красное платье и чёрные туфли
```

Как видим, результат получился верный, хотя аргументы перечислены не в том порядке, что при определении функции. Это происходит потому, что мы явно указали, какие параметры соответствуют каким аргументам.

Следует отметить, что часто программисты на Python путают параметры со значением по умолчанию и вызов функции с именованными аргументами. Это происходит оттого, что синтаксически они плохо различимы. Однако важно знать, что наличие значения по умолчанию не обязывает вас использовать имя параметра при обращении к нему. Также и отсутствие значения по умолчанию не означает, что к параметру нельзя обращаться по имени. Например, для описанной выше функции `Degree` все следующие вызовы будут корректными и приведут к одинаковому результату:

```
>>> Degree(3)
9
>>> Degree(3, 2)
9
>>> Degree(3, a=2)
9
>>> Degree(x=3, a=2)
9
>>> Degree(a=2, x=3)
9
```

Чего нельзя делать, так это ставить обязательные аргументы после необязательных, если имена параметров не указаны:

```
>>> Degree(a=2, 3)
SyntaxError: non-keyword arg after keyword arg
```

8.2.3 Произвольное количество аргументов

Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. Для такого случая есть специальный синтаксис: все параметры обозначаются одним именем (обычно используется имя `args`) и перед ним ставится звёздочка `*`. Например:

```
def unknown(*args):
    for argument in args:
        print (argument)
unknown('Что ', 'происходит', '?')
unknown('Не знаю!')
```

Вывод программы:

```
Что
происходит
?
Не знаю!
```

При этом тип значения `args` — кортеж, содержащий все переданные аргументы по порядку.

8.3 Локальные и глобальные переменные

Если записать в IDLE приведённую ниже функцию, и затем попробовать вывести значения переменных, то обнаружится, что некоторые из них почему-то не существуют:

```
>>> def mathem(a, b):
    a = a/2
    b = b+10
    print(a+b)
>>> num1 = 100
>>> num2 = 12
>>> mathem(num1, num2)
72.0
>>> num1
>>>100
>>> num2
12
>>> a
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
a
NameError: name 'a' is not defined
>>> b
Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
b
NameError: name 'b' is not defined
>>>
```

Переменные `num1` и `num2` не изменили своих первоначальных значений. Дело в том, что в функцию передаются копии значений. Прежние значения из основной ветки программы остались связанны с их переменными.

А вот переменных `a` и `b`, оказывается, нет и в помине (ошибка `name 'b' is not defined` переводится как `"переменная b не определена"`). Эти переменные существуют лишь в момент выполнения функции и называются *локальными*. В

противовес им, переменные num1 и num2 видны не только во внешней ветке, но и внутри функции:

```
>>> def mathem2():
    print(num1+num2)
>>> mathem2()
112
>>>
```

Переменные, определённые в основной ветке программы, являются *глобальными*. Итак, в Python две базовых области видимости переменных:

1. глобальные переменные,
2. локальные переменные.

Переменные, объявленные внутри тела функции, имеют локальную область видимости, те, что объявлены вне какой-либо функции, имеют глобальную область видимости. Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

```
Place = 'Солнечная система' # Глобальная переменная
def global_Position():
    print(Place)
def local_Position():
    Place = 'Земля' # Локальная переменная
    print(Place)
S = input()
if S == 'система':
    global_Position()
else:
    local_Position()
```

Вывод программы при двух последовательных запусках:

```
система
Солнечная система
>>>
========== RESTART: /home/paelius/Python/1.py ==========
планета
Земля
```

Важно помнить, что для того чтобы получить доступ к глобальной переменной на чтение, достаточно лишь указать её имя. Однако если перед нами стоит задача изменить глобальную переменную внутри функции, необходимо использовать ключевое слово global. Например:

```
Number = 10
def change():
    global Number
    Number = 20
print(Number)
change()
print(Number)
```

Вывод программы:

```
10
20
```

Если забыть написать строчку `global Number`, то интерпретатор выдаст следующее:

```
10
10
```

8.4 Программирование сверху вниз

Вряд ли стоило бы уделять много внимания функциям, если бы за ними не скрывались важные и основополагающие идеи. В действительности, функции оказывают решающее влияние на стиль и качество работы программиста. Функция — это не только способ сокращения текста, но что более важно, средство разложения программы на логически связанные, замкнутые компоненты, определяющие её структуру.

Представьте себе программу, содержащую, например, 1000 строк кода (это ещё очень маленькая программа). Обозреть такое количество строк и понять, что делает программа, было бы практически невозможно без функций.

Большие программы строятся методом последовательных уточнений. На первом этапе внимание обращено на глобальные проблемы, и в первом эскизном проекте упускаются из виду многие детали. По мере продвижения процесса создания программы глобальные задачи разбиваются на некоторое число подзадач. Те, в свою очередь, на более мелкие подзадачи и т.д., пока решать каждую подзадачу не станет достаточно просто. Такая декомпозиция и одновременная детализация программы называется *нисходящим методом* программирования или *программированием сверху вниз*.

Концепция функций позволяет выделить отдельную подзадачу как отдельную подпрограмму. Тогда на каждом этапе можно придумать имена функций для подзадач, вписывать в раздел описаний их заголовки и, ещё не добавляя к ним тело функции, уже использовать их вызовы для создания каркаса программы так, будто они уже написаны.

Например, на численных методах студенты решают задачу сравнения двух методов поиска корня уравнения: уже расписанного выше метода деления отрез-

ка пополам и метода Ньютона. Необходимо понять, какой из методов быстрее сходится к корню с заданной точностью.

Концепция программирования «сверху вниз» предусматривает, что вначале студенты напишут «скелет» программы, а уже потом начнут разбираться, как какая функция функционирует в отдельности:

```
def Function(X) #Наше нелинейное уравнение
def Newton(a, b, E)      #Метод Ньютона
def Bisection(a, b, E)   #Метод деления отрезка пополам
A = ?
B = ?
E = ?
Bisection(A, B, E) #Вызов функции метода деления отрезка пополам#
Newton(A, B, E) #Вызов функции метода Ньютона#
```

Какие именно инструкции будут выполняться функциями `Bisection` и `Newton` пока не сказано, но уже известно, что они принимают на вход и что должны возвращать. Это следующий этап написания программы. Потом нам известно, что наше нелинейное уравнение, корень которого необходимо найти, желательно оформить в виде отдельной функции. Так появляется функция `Function`, которая будет вызываться внутри функций `Bisection` и `Newton`.

Когда каркас создан, остаётся только написать тела функций. Преимущество такого подхода в том, что, создавая тело каждой из функций, можно не думать об остальных функциях, сосредоточившись на одной подзадаче. Кроме того, когда каждая из функций имеет понятный смысл, гораздо легче, взглянув на программу, понять, что она делает. Это в свою очередь позволит: (а) допускать меньше логических ошибок и (б) организовать совместную работу нескольких программистов над большой программой.

Важной идеей при таком подходе становится использование локальных переменных. В глобальную переменную можно было бы записать результат работы функции, однако это будет стилистической ошибкой. Весь обмен информацией функция должна вести исключительно через параметры. Такой подход позволяет сделать решение каждой подзадачи максимально независимым от остальных подзадач, упрощает и упорядочивает структуру программы.

8.5 Рекурсивный вызов функции

Рекурсия в программировании — это вызов функции из неё же самой непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция `A` вызывает функцию `B`, а функция `B` — функцию `A`. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

Принцип работы рекурсивной функции удобнее всего объяснять на примере матрёшек (рис. 8.2). Пусть есть набор нераскрашенных матрёшек. Нужно узнать, в какие цвета покрасить самую большую матрёшку. При этом раскрашена только самая маленькая матрёшка, которая не раскрывается. Необходимо последовательно раскрывать матрёшки, пока не дойдём до той, которую раскрыть не получается — это раскрашенная матрёшка. Далее можно раскрашивать каждую следующую матрёшку по возрастанию, держа перед собою предыдущую как пример и затем вкладывать все меньшие, раскрашенные ранее, во вновь раскрашенную.

От матрёшек можно перейти к классическому примеру рекурсии, которым может послужить функция вычисления факториала числа:

```
def fact(num):
    if num == 0:
        return 1
    else:
        return num * fact(num - 1)
n = int(input('Введите число '))
print(fact(n))
```

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду ветвления (выбор одной из двух или более альтернатив в зависимости от условия (условий), которое в данном случае уместно назвать «условием прекращения рекурсии», имеющей две или более альтернативные ветви, из которых хотя бы одна является рекурсивной и хотя бы одна — *терминальной*.

В вышеприведённых примерах с матрёшками и с вычислением факториала условия «если матрёшка не раскрывается» и `if num==0` являются командами ветвления, ветвь `return 1` (или самая маленькая раскрашенная матрёшка) является терминальной, а ветвь `return num*fact(num-1)` (постепенное раскрашивание и закрывание матрёшек) — *рекурсивной*.

Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов — прямой или опосредованный вызов функцией самой себя. Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Помимо функций, выполняющих один рекурсивный вызов в каждой рекурсивной ветви, бывают случаи «параллельной рекурсии», когда на одной рекурсивной ветви делается два или более рекурсивных вызова. Параллельная рекурсия типична при обработке сложных структур данных, таких как деревья. Простейший пример параллельно-рекурсивной функции – вычисление ряда Фибоначчи, где для получения значения n-го члена необходимо вычислить $(n-1)$-й и $(n-2)$-й:

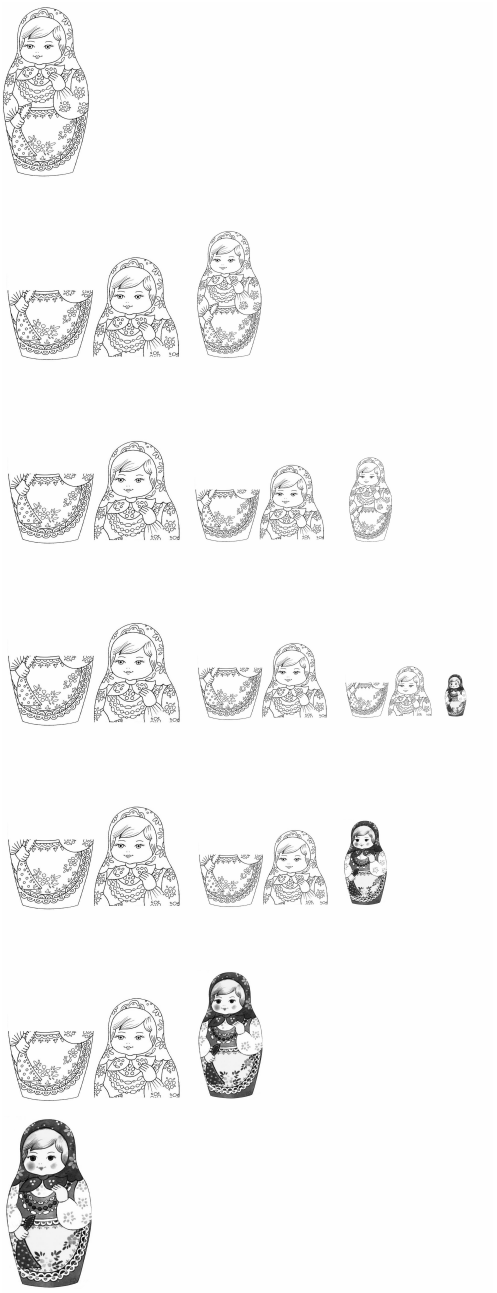


Рис. 8.2. Наглядное представление принципа работы рекурсивной функции.

```
def fib(n):
    if n<3:
        return 1
    return fib(n-1) + fib(n-2)
n = int(input('Введите число: '))
print(fib(10))
```

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов.

Вопрос о желательности использования рекурсивных функций в программировании неоднозначен: с одной стороны, рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм, по сути, рекурсивен. Кроме того, в некоторых декларативных или чисто функциональных языках (таких, как Пролог или Haskell) просто нет синтаксических средств для организации циклов, и рекурсия в них — единственный доступный механизм организации повторяющихся вычислений. С другой стороны, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии. Так, широко распространённый в учебной литературе пример рекурсивного вычисления факториала является, скорее, примером того, как не надо применять рекурсию, так как приводит к достаточно большой глубине рекурсии и имеет очевидную реализацию в виде обычного циклического алгоритма.

8.6 Примеры решения заданий

Пример задачи 32 *Напишите функцию, вычисляющую значения экспоненты по рекуррентной формуле* $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. *Реализуйте контроль точности вычислений с помощью дополнительного параметра* ε *со значением по умолчанию (следует остановить вычисления, когда очередное приближение будет отличаться от предыдущего менее, чем на* 10^{-10}*).*

Реализуйте вызов функции различными способами:

- *с одним позиционным параметром (при этом будет использовано значение по умолчанию);*
- *с двумя позиционными параметрами (значение точности будет передано как второй аргумент);*

- передав значение как именованный параметр.

Решение задачи 32

```
def EXPONENTA(x, eps=10**(-10)):
    ex = 1 # будущий результат
    dx = x # приращение
    i = 2  # номер приращения
    while abs(dx)>eps:
        ex = ex + dx
        dx = dx * x / i
        i = i + 1
    return ex
#Основная программа
A = float(input('Введите показатель экспоненты: '))
print(EXPONENTA(A))
print(EXPONENTA(A, 10**(-4)))
print(EXPONENTA(x = A)
```

Пример задачи 33 Сделайте из функции процедуру (вместо того, чтобы вернуть результат с помощью оператора `return`, выведите его внутри функции с помощью функции `print`).

Решение задачи 33

```
def EXPONENTA(x, eps=10**(-10)):
    ex = 1 # будущий результат
    dx = x # приращение
    i = 2  # номер приращения
    while abs(dx)>eps:
        ex = ex + dx
        dx = dx * x / i
        i = i + 1
    print(ex)
#Основная программа
A = float(input('Введите показатель экспоненты: '))
EXPONENTA(A)
```

8.7 Задания на функции

Задание 31 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Напишите функцию, вычисляющую значения одной из следующих специальных функций по рекуррентной формуле. Реализуйте контроль точности вычислений с помощью дополнительного параметра ε со значением по умолчанию (следует остановить вычисления, когда очередное приближение будет отличаться от предыдущего менее, чем на 10^{-10}). Реализуйте вызов функции различными способами:

- с одним позиционным параметром (при этом будет использовано значение по умолчанию);
- с двумя позиционными параметрами (значение точности будет передано как второй аргумент);
- передав значение как именованный параметр.

Сделайте из функции процедуру (вместо того, чтобы вернуть результат с помощью оператора `return`, выведите его внутри функции с помощью функции `print`).

1. Косинус $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!}$. Формула хорошо работает для $-2\pi \leqslant x \leqslant 2\pi$, поскольку получена разложением в ряд Тейлора возле ноля. Для прочих значений x следует воспользоваться свойствами периодичности косинуса: $\cos(x) = \cos(2 + 2\pi n)$, где n есть любое целое число, тогда `cos(x) = cos(x%(2*math.pi))`. Для проверки использовать функцию `math.cos(x)`.

2. Синус $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$. Формула хорошо работает для $-2\pi \leqslant x \leqslant 2\pi$, поскольку получена разложением в ряд Тейлора возле ноля. Для прочих значений x следует воспользоваться свойствами периодичности косинуса: $\sin(x) = \sin(2 + 2\pi n)$, где n есть любое целое число, тогда `sin(x) = sin(x%(2*math.pi))`. Для проверки использовать функцию `math.sin(x)`.

3. Гиперболический косинус $\mathrm{ch}(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$. Для проверки использовать функцию `math.cosh(x)`.

4. Гиперболический косинус по формуле для экспоненты, оставляя только слагаемые с чётными n. Для проверки использовать функцию `math.cosh(x)`.

5. Гиперболический синус $\mathrm{sh}(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$. Для проверки использовать функцию `math.sinh(x)`.

6. Гиперболический синус по формуле для экспоненты, оставляя только слагаемые с нечётными n. Для проверки использовать функцию `math.sinh(x)`.

7. Натуральный логарифм (формула работает при $0 < x \leqslant 2$):

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{(x-1)^n}{n}$$

Чтобы найти логарифм для $x > 2$, необходимо представить его в виде $\ln(x) = \ln(y \cdot 2^p) = p\ln(2) + \ln(y)$, где $y < 2$, а p натуральное число. Чтобы найти p и y, нужно в цикле делить x на 2 до тех пор, пока результат больше 2. Когда очередной результат деления станет меньше 2, этот результат и есть y, а число делений, за которое он достигнут – это p. Для проверки использовать функцию `math.log(x)`.

8. Гамма-функция $\Gamma(x)$ по формуле Эйлера:

$$\Gamma(x) = \frac{1}{x} \prod_{n=1}^{\infty} \frac{\left(1 + \frac{1}{n}\right)^x}{1 + \frac{x}{n}}$$

Формула справедлива для $x \notin \{0, -1, -2, \dots\}$. Для проверки можно использовать `math.gamma(x)`. Также, поскольку $\Gamma(x+1) = x!$ для натуральных x, то для проверки можно использовать функцию `math.factorial(x)`.

9. Функция ошибок, также известная как интеграл ошибок, интеграл вероятности, или функция Лапласа:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{x}{2n+1} \prod_{i=1}^{n} \frac{-x^2}{i}$$

Для проверки использовать функцию `scipy.special.erf(x)`.

10. Дополнительная функция ошибок:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n)!}{n!(2x)^{2n}}$$

Для проверки использовать функцию `scipy.special.erf(x)`.

Задание 32 (Танцы) *Задания выполняйте все по порядку.*

Составьте функцию, которая по имеющимся именам девочек и мальчиков сделает танцевальные пары девочка–мальчик так, что каждый танцор участвует только в одной паре. Если мальчиков или девочек окажется больше, кто-то из них останется без пары. Выполните один из вариантов, варианты различаются тем, как представлены исходные данные. Дано:

1. два файла с именами в столбик: один — мальчиков, другой — девочек;

2. один файл имён в столбик, на первой строке написано число мальчиков, все имена мальчиков стоят в начале списка;
3. один файл имён в столбик, на первой строке написано число девочек, все имена девочек стоят в конце списка;
4. один файл, написанный в два столбца через пробел: сначала имя ребёнка, потом буква «м» для мальчиков или «ж» для девочек;
5. один файл, написанный в два столбца через пробел: на первой строке написан заголовок в виде двух слов: «мальчики» и «девочки» (заранее неизвестно, какой столбец первый), далее в одном столбце имена мальчиков, во втором — девочек. Один из столбцов может быть длиннее другого, тогда в нескольких последних строчках будет только по одному имени, при этом если первый столбец длиннее, то он начинается с начала строки, если второй, то первым символом перед именем стоит пробел.

Проверьте работу функции на разных примерах:

- когда мальчиков и девочек поровну,
- когда мальчиков больше, чем девочек, и наоборот,
- когда есть ровно 1 мальчик или ровно 1 девочка,
- когда либо мальчиков, либо девочек нет вовсе.

Модифицируйте функцию так, чтобы она принимала второй необязательный параметр — список уже составленных пар, участников которых для составления пар использовать нельзя. В качестве значения по умолчанию для этого аргумента используйте пустой список. Проверьте работу функции, обратившись к ней:

- как и ранее (с одним аргументом), в этом случае результат должен совпасть с ранее полученным;
- передав все аргументы позиционно без имён;
- передав последний аргумент (список уже составленных пар) по имени;
- передав все аргументы по имени в произвольном порядке.

Задание 33 (Создание списков) *Задания выполняйте все по порядку.*

Напишите функцию, принимающую от 1 до 3 параметров — целых чисел (как стандартная функция `range`). Единственный обязательный аргумент — последнее число. Если поданы 2 аргумента, то первый интерпретируется как начальное число, второй — как конечное (не включительно). Если поданы 3 аргумента, то третий аргумент интерпретируется как шаг. Функция должна выдавать один из следующих списков:

1. квадратов чисел;
2. кубов чисел;
3. квадратных корней чисел;
4. логарифмов чисел;
5. чисел последовательности Фибоначчи с номерами в указанных пределах.

Запускайте вашу функцию со всеми возможными вариантами по числу параметров: от 1 до 3.

Подсказка: проблему переменного числа параметров, из которых необязательным является в том числе первый, можно решить 2-мя способами. Во-первых, можно сопоставить всем параметрам нечисловые значения по умолчанию, обычно для этого используют специальное значение `None`. Тогда используя условный оператор можно определить, сколько параметров реально заданы (не равны `None`). В зависимости от этого следует интерпретировать значение первого аргумента как: конец последовательности, если зада только 1 параметр; как начало, если заданы 2 или 3. Во-вторых, можно проделать то же, используя синтаксис функции с произвольным числом параметров; в таком случае задавать значения по умолчанию не нужно, а полезно использовать стандартную функцию `len`, которая выдаст количество реально используемых параметров.

Задание 34 (Интернет-магазин) *Задания выполняйте все по порядку.*

Решите задачу об интернет-торговле. Несколько покупателей в течении года делали покупки в интернет-магазине. При каждой покупке фиксировались имя покупателя (строка) и потраченная сумма (действительное число). Напишите функцию, рассчитывающую для каждого покупателя и выдающую в виде словаря по всем покупателям (вида имя:значение) один из следующих параметров:

1. число покупок;
2. среднюю сумму покупки;
3. максимальную сумму покупки;
4. минимальную сумму покупки;
5. общую сумму всех покупок.

На вход функции передаётся:

- либо 2 списка, в первом из которых имена покупателей (могут повторяться), во втором – суммы покупок;
- либо 1 список, состоящий из пар вида `(имя, сумма)`;
- либо словарь, в котором в качестве ключей используются имена, а в качестве значений — списки с суммами.

Глава 9

Модули

Модульное программирование — метод создания программного обеспечения, который подчёркивает разделение функциональности программы на независимые взаимозаменяемые блоки, называемые модулями.

Модуль — функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом. Модули проектируются таким образом, чтобы предоставлять программистам удобную для многократного использования функциональность в виде набора функций, классов, констант. Модули могут объединяться в пакеты и, далее, в библиотеки. Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы.

Модульное программирование тесно связано со структурным и объектно-ориентированным программированием, все они имеют одну и ту же цель — облегчить построение больших программ путём декомпозиции на более мелкие части. Все они возникли примерно в 1960-х годах. В настоящее время эти понятия имеют немного разный смысл: модульное программирование теперь относится к высокоуровневой декомпозиции кода всей программы на части; структурное программирование — к низкоуровневому синтезу программы из структурированных блоков; объектно-ориентированное программирование — к высокоуровневому синтезу программы из объектов, включающих как функции/методы, так и сами данные.

Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Модульность часто является средством упрощения задачи проектирования программы и распределения процесса разработки между группами разработчиков. При разбиении программы на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.

Следующие языки поддерживают парадигму модульного программирования (список неполный): Ada, Lisp, D, Erlang, F#, Fortran, Go, Haskell, MATLAB, ML (Standard ML и OCaml), Modula (1, 2, 3), Oberon (1, 2), Pascal (практически все

версии), Perl, Python (2 и 3), Ruby, Rust. Во многих других языках также возможно написание программы в виде нескольких отдельных файлов, например, в C, C++, Java, C# и ряде других, однако используемый в них подход нельзя полностью назвать модульным, поскольку отсутствует либо возможность раздельной компиляции модулей, либо возможность писать в модуле произвольные локальные и глобальные переменные (подход Java, когда файл — это класс), в том числе разделять переменные и функции на доступные и недоступные внешним программам (в Java и её наследниках это реализуется за счёт классов).

Любая программа на Python может считаться модулем (да-да, все те программы, которые вы писали, можно назвать модулями). Это отличает Python от Pascal и его наследников, где модуль и главная программа оформляются немного различно. В этой главе упорядочим все способы подключения стандартных модулей для Python, а также научимся создавать свои модули и подключать их.

Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на C или C++.

9.1 Подключение стандартного модуля

Вначале нам необходимо упорядочить все свои знания о подключении стандартных модулей, полученные в результате чтения и выполнения заданий из Части I.

Первый способ подключить модуль: с помощью инструкции **`import`**.

После ключевого слова **`import`** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода:

```
import math
import numpy, scipy
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе `e`, расположенной в модуле `math`:

```
math.e
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение `AttributeError`. А если не удастся найти модуль для импортирования, то `ImportError`.

Второй способ подключить модуль: использовать псевдонимы.

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова `as`.

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

Теперь доступ ко всем атрибутам модуля numpy осуществляется только с помощью переменной `np`, а переменной `numpy` в этой программе уже не будет:

```
np.dot(a, b)
```

Третий способ подключить модуль: с помощью инструкции **`from`** (первый формат).

Подключить определенные атрибуты модуля можно с помощью инструкции **`from`**. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> as <Псевдоним 1>,
    <Атрибут 2> as <Псевдоним 2>, ... ]
```

Конкретный пример:

```
from random import uniform as uni, normalvariate as norm,
    expovariate as exp
```

Этот формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова `as`.

```
>> from math import e, ceil as c
>> e
2.718281828459045
>> c(4.6)
5
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

```
>> from math import (sin, cos,
...           tan, atan)
```

Четвёртый способ подключить модуль: с помощью инструкции **`from`** (второй формат).

```
from <Название модуля> import *
```

Второй формат инструкции **`from`** позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все константы из подмодуля `constants` из модуля `scipy`:

```
>> from scipy.constants import *
>> e # заряд электрона
1.6021766208e-19
>> pi # число пи
3.141592653589793
>> g # ускорение свободного паданения
9.80665
```

```
>> Avogadro # число Авогадро
6.022140857e+23
>> m_e # масса электрона
9.10938356e-31
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная `__all__` (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная `__all__` не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имён главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

9.2 Создание и подключение собственного модуля

Если вы честно прочитали первую часть книги целиком, а самое главное, честно выполнили все задания в ней, то у вас уже должна сформироваться естественная потребность в выделении написанных вами лично полезных функций в отдельный файл-модуль. Вначале вам было страшно писать даже простые линейные программы. Потом для решения более сложных задач вам стало не хватать линейных алгоритмов и вы освоили циклы и ветвления. Затем вы заметили, что прописываете в цикле подряд одни и те же действия, но с разными значениями переменных; так вы последовательно пришли к выделению таких кусков в отдельные функции с переменными параметрами. И, наконец, все полезные, уже отлаженные функции есть смысл выделить в отдельный файл, в который не стоит лазить без крайней необходимости. Почему? Потому что вы оттестировали каждую лежащую там функцию на большом количестве примеров, и если при использовании этой функции в вашей новой программе возникла ошибка, то вы можете быть уверенны в том, что ошибка где-то в новом коде, а значит, вам не нужно проверять тот кусок кода, который лежит в вашем модуле.

Теперь пришло время создать свой модуль. Как же его назвать? Помните, что вы (или другие люди) будут его импортировать и использовать в качестве переменной. Название модуля не может содержать пробел. Если вам очень хочется назвать модуль `module DaNya`, то используйте символ нижнего подчёркивания `module_DaNya`. Модуль нельзя именовать также, как и ключевое слово. Список ключевых слов: `False` — ложь, `True` — правда, **`and`** — логическое И, **`or`** — логическое ИЛИ, **`not`** — логическое НЕ, **`in`** — проверка на вхождение, **`if`** — если, **`else`** — иначе, **`elif`** — в противном случае, если, **`for`** — цикл for, **`while`** — цикл while, `with/as` — менеджер контекста, `assert` — возбуждает исключение, если условие ложно, **`break`** — выход из цикла, **`class`** — пользовательский тип, состоящий из методов и атрибутов, **`continue`** — переход на следующую итерацию цикла, **`def`** — определение функции, **`del`** — удаление объекта, **`except`** — перехватить исключение, **`finally`** — вкупе с инструкцией **`try`**, выполняет инструкции независимо

от того, было ли исключение или нет, **from** — импорт нескольких функций из модуля, **global** — позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции, **import** — импорт модуля, **is** — ссылаются ли 2 объекта на одно и то же место в памяти, **lambda** — определение анонимной функции, **nonlocal** — позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции, **pass** — ничего не делающая конструкция, **raise** — возбудить исключение, **return** — вернуть результат, **try** — выполнить инструкции, перехватывая исключения.

Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций (типа **type, float, int**, **abs, input, print**, т.е. те, которые в IDLE подсвечиваются фиолетовым цветом). То есть, конечно, можно, но это создаст большие неудобства при его последующем использовании.

Создадим файл с разрешённым и более-менее осознанным названием `Methods.py`, в котором определим две функции поиска корня уравнения: методом деления отрезка пополам `Bisection` и методом Ньютона `Newton`. Помните, мы создавали эти функции в предыдущей главе?

```
from math import *
def function(x): #уравнение, которое нужно решить
    f = x**2-4
    return f
def derivative(x): #производная
    f = 2*x
    return f
def Newton(a, b, e):
    x0 = (a+b) / 2
    delta = function(x0)/derivative(x0)
    n = 0
    while abs(delta) > e:
        n = n + 1
        delta = function(x0)/derivative(x0)
        x0 = x0 - delta
    return (x0, n)
def Bisection(a, b, e):
    n = 0
    while (b-a) > e:
        n = n + 1
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
```

```
    return ((a+b)/2, n)
```

Теперь в этой же папке создадим другой файл, например, `Main.py`:

```
import Methods
A = -10
B = 8
E = 10**(-15)
print(Methods.Bisection(A, B, E))
print(Methods.Newton(A, B, E))
```

Выведет:

```
(-2.0, 54)
(-2.0, 7)
```

Поздравляю! Вы сделали свой модуль!

Но у нашей программы есть существенный недостаток — нелинейное уравнение, которое нужно решить, задаётся внутри модуля. Значит, если нам нужно сменить уравнение, придётся менять модуль. В идеале модуль должен быть изолирован от конечного пользователя. Пользователь может использовать определённые в нём функции и переменные, но не должен для этого менять сам код модуля — править файл. Чтобы достичь этого, будем передавать наше уравнение как параметр функций `Bisection` и `Newton`. Тогда основная программа `Main` будет выглядеть следующим образом:

```
import Methods
def func(x):
    f = x**2-4
    return f
A = -10
B = 8
E = 10**(-15)
print(Methods.Bisection(A, B, E, func))
print(Methods.Newton(A, B, E, func))
```

Но теперь встаёт вопрос, что делать с вычислением производной. Модифицируем модуль `Methods`, применив численное дифференцирование. Производная функции определяется выражением:

$$f'(x_0) = \frac{df}{dx} = \lim_{dx \to 0} \frac{f(x_0 + dx) - f(x_0)}{dx}. \tag{9.1}$$

Заменяя приращение dx на конечную величину Δx, называемую шагом дифференцирования, методом односторонней разности справа получаем выражение:

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}. \tag{9.2}$$

Если дифференцируемая функция задана уравнением, то для вычисления значения дифференциала необходимо получить значение функции $f(x)$ в точке x_0 и в точке $x_0 + \Delta x$. После чего можно вычислить значение производной функции $f'(x_0)$.

Нетрудно записать выражение для левосторонней разности:

$$f'(x_0) = \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x}. \tag{9.3}$$

С точки зрения точности оба эти подхода равнозначны. Более точное значение даёт метод двусторонней разности (что особенно справедливо для гладких функций):

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x}. \tag{9.4}$$

Тогда в модуле `Methods` получится:

```
from math import *
def derivative(x, function):
    dx = 10**(-10) #шаг дифференцирования
    f = (function(x+dx)-function(x-dx))/(2*dx)
    return f
def Newton(a, b, e, function):
    x0 = (a+b)/2
    delta = function(x0)/derivative(x0, function)
    n = 0
    while abs(delta) > e:
        n = n + 1
        delta = function(x0)/derivative(x0, function)
        x0 = x0 - delta
    return (x0, n)
def Bisection(a, b, e, function):
    n = 0
    while (b-a) > e:
        n = n + 1
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
    return ((a+b)/2, n)
```

Когда модуль написан, прежде, чем делать его полностью независимым блоком программы, всегда необходимо протестировать его. Поэтому некоторое время он побудет у нас самостоятельной программой. В этот момент вы будете выступать уже не в роли пользователя, а в роли самого настоящего разработчика и

тестировщика. Поэтому на вас накладывается дополнительная ответственность, вы теперь обязаны думать не только о себе, но и о конечном пользователе.

Итак, вам необходимо помнить, что при импортировании модуля его код выполняется полностью, то есть, если содержится не только определение функций, но и какие-то действия, которые были необходимы вам для процесса тестирования: вычисления, программа что-то печатает, рисует, сохраняет, то при её импортировании это будет выполнено. Этого можно избежать, если проверять, запущен ли скрипт как программа, или импортирован. Для этого существует переменная `__name__` — обратите внимание на два подчёркивания в начале, так обозначают специальные, «магические» переменные, часто они хранят состояние интерпретатора. Если наш файл с кодом запущен из операционной системы или среды разработки сам по себе, то есть он есть главная программа, переменная `__name__` равна `'__main__'`. Если же он импортирован другим файлом программы, при выполнении кода данного файла переменная `__name__` будет равна имени файла без расширения `py`. Например, в наш модуль для решения нелинейных уравнений можно добавить в конец следущий код:

```
if __name__ == '__main__':
    import matplotlib.pyplot as plt
    def testfunc(x):
        return (x-1)*(x-10)
    for epspok in range(2, 22): #показатели двойки
        eps = 2**-epspok
        x, n = Bisection(-4, 2, eps, testfunc)
        plt.plot(epspok, n, '.', color = 'black')
    plt.show()
```

Этот код позволит протестировать нашу функцию `Bisection` на квадратном уравнении, имеющем два корня: 1 и 10, запуская модуль как самостоятельную программу. Сначала мы печатаем результат вызова функции `Bisection` с разными значениями начала и конца отрезка, чтобы убедиться, что результат правильный и оба корня находятся. Затем мы вызываем функцию в цикле при разных значениях точности, чтобы проверить зависимость числа шагов от требуемой точности. Теоретически зависимость должна быть логарифмическая: $n \approx -log_2(\varepsilon)$, где n — число шагов, а ε — требуемая точность (в программе обозначена `eps`). Поэтому по оси абсцисс мы задаём и откладываем не сами значения точности `eps`, а двоичных логарифм от них `epspok`. Важно, что этот тестовый код будет вызываться только, когда наш модуль — главная программа, а если он подключается из другой программы, наши проверки не будут работать и не станут мусорить в вывод и рисовать неуместный график.

Теперь ваш модуль протестирован и полностью готов к автономной работе. Встаёт вопрос, куда же его поместить? Чтобы интерпретатор мог импортировать ваш модуль, он должен понимать, где модуль находится. Поместить его просто где-то в операционной системе недостаточно. Модуль должен лежать либо в той же папке, что и импортирующая его программа, либо по одному из адресов,

указанных в списке `sys.path` (фактически, это просто список из строк, ничего специфического). Чтобы посмотреть, что за пути прописаны по-умолчанию (при старте интерпретатора), нужно импортировать модуль `sys` и затем напечатать содержимое списка `sys.path`:

```
import sys
for d in sys.path:
    print(d)
```

В нашем случае получилось вот что (результат может зависеть от того, каким конкретно образом установлен Python, сколько дополнительных модулей вы ставили и какими средствами):

```
C:\Python3\WPy64-3810\notebooks
C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages\idlexlib\idlefork
C:\Python3\WPy64-3810\python-3.8.1\scripts
C:\Python3\WPy64-3810\python-3.8.1\python38.zip
C:\Python3\WPy64-3810\python-3.8.1\DLLs
C:\Python3\WPy64-3810\python-3.8.1\lib
C:\Python3\WPy64-3810\python-3.8.1
C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages
C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages\win32
C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages\win32\lib
C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages\Pythonwin
```

Если вы не хотите таскать ваш модуль вслед за вызывающей его программой, а обеспечить доступ к нему на постоянной основе, можно положить его в одну из этих папок. В качестве комментария к указанному выше списку можно сказать, что папка `C:\Python3\WPy64-3810\python-3.8.1\lib` ведёт к некоторым модулям стандартной библиотеки — тем, что написаны на самом Python, а не реализованы в виде расширений на других языках (там лежит, в частности, модуль `os`), а папка `C:\Python3\WPy64-3810\python-3.8.1\lib\site-packages` ведёт к сторонним модулям (именно там лежат `numpy` и `matplotlib`).

Есть альтернативный подход: не помещать модули туда, куда указывает `sys.path`, а добавить нужный вам путь в сам `sys.path`, поскольку это всего лишь список, а список можно модифицировать. Таким образом можно будет положить модуль в любое удобное для вас место. Например, вы решили, что все модули должны лежать в отдельной папке `units`, находящейся там же, где и главная программа. В таком случае без дополнительных действий с вашей стороны Python их уже не найдёт и выдаст ошибку. Для того, чтобы можно было импортировать наш модуль, придётся в главной программе дописать две строки:

```
import sys
sys.path.append('units')
```

Это обеспечит доступ разом ко всем модулям, лежащим в папке `units`. Причём сделать это нужно до того, как импортировать наши модули или отдельные

функции и значения из них. Правда, работать нормально такой подход будет только, если запускать программу из того же каталога, где она расположена, иначе модули будут искаться в подкаталоге `units` каталога, откуда запущена программа, а это совсем не то, чего мы хотели бы. Чтобы решить проблему толково, нужно уметь вычислять путь к самому главному файлу программы. Для этого в Python есть инструменты, хотя они более мудрёные, к тому же придётся задействовать три стандартных модуля: `sys`, `os.path` (подмодуль модуля `os`) и `inspect` — это модуль, помогающий отслеживать актуальное состояние интерпретатора, в том числе пути, загруженные модули, классы, объекты и др. Из последнего модуля нам нужны будут две функции: `currentframe`, возвращающая специальный объект типа `frame`, и `getsourcefile`, которая по этому объекту может определить, какой файл ему соответствует. Ниже — исправленный пример.

```
from inspect import getsourcefile, currentframe
from os.path import split, abspath, join
import sys

weg = split(abspath(getsourcefile(currentframe())))[0]
sys.path.append(join(weg, 'units'))
```

9.3 Задания на работу с модулями

Задание 35 *Задания выполняйте все по порядку.*

Напишите модуль, реализующий описанные выше методы решения нелинейных уравнений. Далее, используя конструкцию с `__name__ == '__main__'`, протестируйте оба метода на точность внутри данного модуля.

Задание 36 *Задания выполняйте все по порядку.*

Положите написанный вами в предыдущем задании модуль в специально созданную папку `modules`. Напишите программу, которая с использованием написанного ранее модуля решает на отрезке $[0, \pi]$ следующие нелинейные уравнения и строит графики зависимости числа шагов n от требуемой точности ε (точность можно менять в диапазоне от 2^{-2} до 2^{-22}, по оси абсцисс откладывается модуль показателя степени двойки). Расчёты сделайте методом деления отрезка пополам и методом Ньютона. Не забудьте вывести на график легенду `legend()`.

1. $\sin\left(x - \frac{\pi}{6}\right) - 0.5 = 0$, должно получиться $x_{true} = \frac{\pi}{3}$;
2. $\cos(x) - 0.5 = 0$, должно получиться $x_{true} = \frac{\pi}{3}$;
3. $\mathrm{tg}(x) = 1$, должно получиться $x_{true} = \frac{\pi}{4}$;
4. $\arctan(x) = \frac{\pi}{3}$, должно получиться $x_{true} = \sqrt{3}$;

5. $(x+4)(x-1)(x-20)(x+33)=0$, должно получиться $x_{true}=1$.

Задание 37 Задания выполняйте все по порядку.

Постройте зависимость разности между численным и аналитическим решениями $(x-x_{true})$ от числа шагов n. На одном графике постройте и сравните эти зависимости при расчётах методом деления отрезка пополам и методом Ньютона. Не забудьте вывести на график легенду `legend()`.

Глава 10

Функциональное программирование

Выше мы уже разобрали несколько подходов (часто их называют «парадигмами») к программированию: структурный — самый хронологически ранний подход, состоящий в выделении составных операторов, содержащих в себе другие операторы (циклов, ветвлений), а также автономных блоков кода, называемых подпрограммами, или функциями; модульный — продолжение и расширение структурного; объектно-ориентированный, представляющий дальнейшее развитие модульного, когда структуры данных и код группируются не в модулях, а в классах. Ещё одна распространённая парадигма — функциональное программирование. Под функциональным программированием в литературе понимается совокупность нескольких отдельных идей и их реализаций, связанных между собою, но не представляющих органичное целое. Часто функциональное программирование противопоставляют императивному (всё ранее рассмотренное программирование, включая объектно-ориентированное, можно считать императивным). Такое противопоставление основано на понимании императивного программирования как набора команд процессору, что делать по шагам, а функционального — как описания решения задачи в виде функций (формул). В действительности, чисто функциональное программирование практически невозможно, даже самый чистый из функциональных языков — Haskell — имеет специализированный механизм монад для обеспечения связи со внешним миром, например, чтения файлов, работы с сетью и устройствами ввода-вывода.

Функциональное программирование *не нужно*, если требуется просто посчитать результат по какой-нибудь сложной формуле, например, для решения кубического уравнения или вычисления объёма полого конуса, никакие списки не нужны. Правда, в таком случае объектно-ориентированное или структурное программирование тоже не нужно. Честно говоря, не нужны в таком случае даже циклы, если только они не используются для разложения какой-нибудь трансцендентной функции в ряд. Поэтому такие задачи мы рассматривать не будем.

Когда заводят речь о функциональном программировании, много говорят о неизменности данных (идею неизменности данных в Python удачно иллюстри-

руют строки и кортежи), чистых функциях (таких функциях, которые не используют никакие глобальные и нелокальные переменные и не модифицируют свои аргументы), функциях высших порядков (обычно встречается триада **`map`**, **`filter`**, **`reduce`**, но бывают и другие), безымянных функциях (их ещё называют λ-функциями) и др. — все они есть в Python. Но с практической точки зрения первое, что увидит перед собою программист, переходящий с императивной на функциональную парадигму, — это огромное число циклов, которые нужно чем-то заменить, потому что *в функциональном программировании не бывает циклов*! И практически вся его работа изначально будет сводиться к замене циклов рекурсией. В действительности, можно писать функциональный код без многих атрибутов функционального программирования, пусть он будет громоздким, но нельзя целиком обойтись без рекурсии.

10.1 Списки и рекурсия

Один из первых функциональных языков программирования — Lisp, название которого переводится как язык обработки списков. При его создании была заложена довольно простая идея: большинство сложных вычислений, которым требуются повторяющиеся операции, можно организовать через списки. В самом деле, большинство практических задач можно свести к одному из следующих вариантов:

1. создание некоторого списка на основе правила, формулы, алгоритма, в том числе при помощи других, меньших списков, например, списка чисел Фибоначчи или списка разрешённых правилами автомобильных номеров на основе списков букв и цифр;

2. преобразование списка в другой список, например, если нам нужно протабулировать некоторую функцию, тот же синус, чтобы потом нарисовать график;

3. свёртка списка — преобразование его к одному значению, например, вычисление суммы, среднего, максимального или минимального элемента, длины и др.

В действительности, некоторые такие операции не требуют обязательного создания списка, например, вычисление приближенного значения для оценки золотого сечения или числа π по рекуррентной формуле не предполагает обязательного хранения всех промежуточных вычислений. Но в таком случае нам пришлось бы пожертвовать неизменностью данных (переписывать текущее приближение на каждом шаге), что при функциональном подходе недопустимо. Поэтому фактически задача всё равно сводится к созданию списка, пусть даже от него нам нужен только последний элемент.

В императивном программировании для решения всех трёх типов задач используются циклы, в функциональном — рекурсия и функции высших порядков.

При этом надо понимать, что рекурсия в принципе способна решать некоторые типы задач, которые с помощью циклов, даже вложенных, решить нельзя.

Рассмотрим первую из трёх обозначенных выше задач — задачу о создании списка. Произведём расчёт дигамма-функции целого аргумента. Дигамма-функция, обозначаемая обычно $\psi(x) = \frac{d}{dx}\ln\big(\Gamma(x)\big)$, довольно востребованная трансцендентная функция, используемая, например, для расчёта некоторых мер теории информации: взаимной информации и энтропии переноса по формулам, выведенным в работе Козаченко–Леоненко. Эти меры популярны в ряде приложений, в том числе при изучении мозга.

Для решения задачи можно использовать следующее рекуррентное соотношение:

$$\psi(x+1) = \psi(x) + \frac{1}{x}, \tag{10.1}$$

где $\psi(1) = -\gamma$, $\gamma \approx 0.57721566490153286060$ — константа Эйлера–Маскерони.

С помощью цикла вычислить $\psi(n)$, где n натуральное число, довольно легко:

```
n = int (input ('Введите натуральное число '))
psi = -0.57721566490153286060
for i in range(1, n):
    psi = psi + 1/i
print(psi)
```

При этом нет нужды создавать какие-либо функции. Но если мы хотим заменить цикл на рекурсию, *функцию придётся создать обязательно*! Вот вариант с рекурсией вместо цикла:

```
def digamma(psii, n):
    i = len(psii)
    if i < n:
        psi = psii[-1] + 1/i
        return digamma(psii + [psi], n)
    else:
        return psii
psivals = digamma([-0.57721566490153286060], n)
print(psivals[-1])
```

Как и положено в рекурсивных функциях обязательно присутствуют две ветви: рекурсивная и нерекурсивная. Рекурсивная вычисляет новый элемент, при этом все промежуточные значения складываются в список, который увеличивается на один новый элемент на каждом запуске, для чего используется оператор сложения списков `psii + [psi]` и передаётся этой же функции при очередном вызове. Нерекурсивная возвращает результат в виде списка всех значений дигамма-функции от аргументов, начиная с 1 и до n. Предложенный вариант берёт номер итерации `i` путём вычисления длины списка на каждом шаге, иначе его пришлось бы передавать в качестве третьего аргумента функции `digamma`.

Представленная реализация — чистая функция, потому что она не меняет свои входные аргументы (на каждом шаге создаётся новый список, а не модифицируется исходный) и не обращается ни к каким внешним переменным. Выглядит такая реализация, однако, несколько несуразно: при вводе приходится передавать начальное значение, а при выводе результата приходится брать последний элемент списка. К тому же создание списков на каждом шаге — очень плохой подход с точки зрения расходования памяти, поскольку в таком случае на каждом шаге памяти нужно всё больше и больше, и общий её израсходованный объём будет порядка $O(n^2)$, то есть пропорционален квадрату n, что совсем никуда не годится. Поэтому такие чистые реализации не очень распространены на практике. Чаще используют приём, известный как «замыкание», когда рекурсивная функция вкладывается внутрь другой, нерекурсивной, обращаясь к локальным переменным внешней функции. Вероятно, смысл термина «замыкание» в том, что внутренняя функция замкнута в наружную, в неё же замкнуты все нелокальные переменные, так что наружная функция получается в итоге чистая, хотя внутренняя — нет. Вот как можно написать пример выше с использованием замыкания и даже без списка:

```
def digamma2(n):
    def dgf(i, psi):
        if i < n:
            return dgf(i+1, psi+1/i)
        else:
            return psi
    return dgf(1, -0.57721566490153286060)
print(digamma2(n))
```

Обратите внимание, мы спрятали начальное значение `-0.57721566490153286060` внутрь внешней функции `digamma2` как литерал, используемый при первом вызове, — теперь его не нужно указывать при обращении, а в теле внутренней функции `dgf` обращаемся к переменной `n`, для неё внешней. Но настоящих глобальных переменных тут нет. Более того, принцип неизменности данных сохраняется: мы не меняем значения никаких переменных. В некоторых языках программирования, где функции внутри функций не разрешены, замыкания невозможны. Например, в C и C++.

Если нам всё же нужен весь список значений дигамма-функции от аргументов от 1 до n включительно, например, для целей вычисления взаимной информации, его можно получить путём следующей модификации последнего варианта программы:

```
def digamma3(n):
    dglist = [-0.57721566490153286060]
    def dgf(i):
        dglist.append(dglist[-1]+1/i)
        if i < n-1:
            dgf(i+1)
```

```
    dgf(1)
    return dglist
print(digamma3(n))
```

Теперь на верхнем уровне мы ввели список `dglist`, который доступен вложенной функции `dgf`, как и величина `n`, а функция `dgf` выродилась по сути в процедуру, поскольку оператор `return` больше не нужен. Более того, исчезла даже нерекурсивная ветка в условии, что кажется прямым нарушением принципов построения рекурсивных функций. На самом деле она есть, просто Python позволяет не писать `return None`, а это всё, что есть в нерекурсивной ветке. Фактически, вся работа рекурсивной функции сводится к модификации списка `dglist`. Такой код ещё дальше от чисто функционального стиля за счёт введения изменяемого списка, но требование чистоты внешней функции соблюдается, так как `dglist` — её локальная переменная, а не формальный параметр.

10.2 Списки и функции высших порядков

Теперь рассмотрим вторую задачу из изложенных выше — о построении одного списка на основе другого, уже существующего. Функциональный подход предлагает два варианта решения таких задач. Первый, уже рассмотренный выше, — через рекурсию. Второй — с помощью функций высшего порядка **`map`** и **`filter`**.

10.2.1 Преобразование списков с помощью `map`

Для иллюстрации работы функции **`map`** часто рассматривают задачу о построении параболы или синуса, для чего нужно по списку значений аргумента получить значения функции для каждой из них, но мы для разнообразия отойдём от числовых данных и рассмотрим текстовые. Из списка мужских имён, заканчивающихся на твёрдый согласный, создайте список фамилий, образованных от них путём добавления суффикса «-ов».

Первое решение — с использованием цикла **`for`**:

```
imena = ['Агафон', 'Богдан', 'Варлам', 'Герасим', 'Демьян', 'Емельян']
familii_c = []
for ime in imena:
    familii_c.append(ime+'ов')
print(familii_c)
```

Второе решение — с использованием рекурсивной функции:

```
def ov(imena, familii):
    if len(familii) < len(imena):
        return ov(imena, familii + [imena[len(familii)]+'ов'])
    else:
        return familii
```

```
familii_ov = ov(imena, [])
print(familii_ov)
```

Третье решение — с использованием функции **map**:

```
def fam(ime):
    return ime+'ов'
familii_m = list(map(fam, imena))
print(familii_m)
```

Прокомментируем наши решения. Первое — с использованием цикла — должно быть уже хорошо понятно. Мы так делали много раз: сначала создаётся пустой список и потом в него с помощью встроенного метода `append` по определённому правилу добавляются значения. Второе решение тоже должно быть понятно: тут рекурсивная функция получает на вход два списка: первый остаётся неизменным (это входной список), второй либо наращивается в рекурсивной ветке одним значением (это список–результат), если он короче первого, либо выдаётся в качестве результата в нерекурсивной ветке, если их длины совпали. При внешнем вызове рекурсивной функции второй список задаётся пустым по аналогии с тем, что мы видели при использовании цикла. То есть рекурсия здесь подменяет собою цикл как и в ранее рассмотренных примерах. Третье решение для нас — принципиально новое. Здесь используется функция **map**, часто называемая функцией высшего порядка, потому что одним из её аргументов является другая функция, а вторым — перечисляемый объект (как правило, список). Функция **map** применяет свой первый аргумент — функцию — ко всем *элементам* второго аргумента по порядку и выдаёт новый перечисляемый объект — последовательность возвратов применённой функции. Никакого явного цикла здесь нет и рекурсии тоже. Фактически, функция **map** для нас — это новая абстракция, ничего подобного мы ранее не видели. Ближайший аналог из ранее рассмотренного — встроенные функции `numpy`, которые тоже принимали массив и выдавали массив. Только там они сами обозначали собою, что они делают, например, `numpy.sin` считала синус. А **map** ничего сама не делает, она без первого аргумента — пустая оболочка. Прямой полной аналогии с циклом здесь нельзя достичь; Python, используя специальный оператор–затычку **pass**, позволяет создать цикл, который ничего не делает:

```
for ime in imena:
    pass
```

Такой цикл действительно проработает столько раз, сколько элементов в списке `imena`, ничего не делая на каждом шаге. Но в случае с **map** указание применяемой функции — обязательно, а вот сам перечисляемый объект, к элементам которого применяется эта функция, может быть и пустым (аналогично тому, что цикл не проработает ни разу).

Отметим, что на самом деле все решения имеют определённые черты, выдающие особенности языка Python. Решение через цикл использует обход списка

по значению, что в «старых» языках типа C или Pascal невозможно, возможен только обход по номеру. В более современных языках такой цикл есть и часто называется `foreach`. Вариант с рекурсией опирается на использование встроенной функции длины, чтобы определить, достиг ли список фамилий длины списка имён. Опять же в старых языках с этим есть проблемы: в C массивы (реальных динамических массивов там нет, есть только указатели) своей длины не знают и определить её встроенными методами невозможно, а в классическом Pascal нет массивов изменяющейся длины. Вариант с функцией **`map`** требует явного преобразования результата к списку, так как **`map`** выдаёт итератор и физически объект не формирует. Это даёт преимущества при использовании так называемого *конвейера* из функций. К тому же результат можно преобразовать не к списку, а к кортежу, например. Во многих других языках **`map`** будет выдавать сам список, так было и в Python 2.

10.2.2 Безымянные функции

В представленном нами случае используемая в **`map`** функция `fam` очень простая и выделение её в отдельную функцию явно выглядит избыточно и в этом смысле вариант с циклом имеет явное преимущество, а решение в функциональном стиле всё ещё выглядит избыточно громоздко, хотя уже лучше, чем через рекурсию. Чтобы уменьшить объём кода и не плодить лишние, впоследствии никогда не используемые сущности, Python поддерживает механизм безымянных (анонимных) однострочных функций, также известных как **`lambda`**-функции или просто «лямбдами». Такие функции в обязательном порядке есть во многих функциональных языках, а недавно их добавили даже в стандартны Java и C++, просто потому что в той же Java обычные функции в принципе отсутствуют, а могут быть только методами класса, а создавать класс, чтобы добавить пару символов к строке или сложить два числа, — *Яв*ный перебор. Вот решение нашей предыдущей задачи с использованием **`lambda`**-функции:

```
familii_l = list(map(lambda v: v+'ов', imena))
print(familii_l)
```

Как видим, фактическое решение уложилось теперь в одну строку и стало уже заметно короче того, что можно сделать с использованием цикла.

Синтаксис у безымянной функции такой: сначала идёт ключевое слово **`lambda`**, обозначающее, что далее — безымянная функция, затем через запятую идёт перечисление параметров, в нашем случае параметр один `v` (обратите внимание: скобок нет!), потом после двоеточия следует результат, то есть то, что будет стоять в нормальной функции после ключевого слова **`return`**. Вся запись, начиная от ключевого слова **`lambda`** и заканчивая результатом, стоит на том месте, где при использования обычной функции расположен единственный идентификатор — её имя, в нашем случае — до запятой, отделяющей первый аргумент **`map`** (применяемую функцию), от второго аргумента — списка, к элементам которого эта функция применяется. Такой стиль практически копирует

то, как функции объявляются в функциональных языках, например, семейства ML, где скобки вокруг аргументов отсутствуют и после разделителя заголовка и тела приводится результат (формула) вычисления. Результат обязан быть записан как один оператор. Это значит, что никаких локальных переменных (не путайте с формальными параметрами, то есть со значениями, стоящими в заголовке, — аргументами функции, они как раз могут быть, в представленном случае это переменная `v`, которую мы тут ввели) в безымянной функции не может быть создано, запуск методов, изменяющих что-то на месте типа методов списка **`append`** или **`sort`**, невозможен по той же причине: методы представляют собою отдельные операторы, возвращающие `None`, а его некуда девать. Это полностью согласуется с идеологией функционального программирования. Впрочем, если внутри функции нужно сделать что-то сложное, никто по-прежнему не запрещает описать её как отдельную «большую» функцию с именем и воспользоваться всеми стандартными возможностями.

Для того, чтобы увеличить возможности использования функционального подхода и безымянных функций, в Python введён дополнительный однострочный вариант оператора **`if`**/**`else`** с обеими ветвями. Для его иллюстрации рассмотрим немного усложнённый вариант предыдущей задачи. Из списка мужских имён, заканчивающихся на твёрдый согласный или на «-ей/-ой» (например, «Андрей», «Сысой»), создайте список фамилий, образованных от них путём добавления суффиксов «-ов/-ев».

Первое решение — с использованием именованной функции:

```
imena2 = ['Алексей', 'Богдан', 'Веденей', 'Герасим', 'Дорофей', 'Емельян']
def fam2(ime):
    if ime.endswith('ей') or ime.endswith('ой'):
        return ime[:-1] + 'ев'
    else:
        return ime + 'ов'
familii_m2 = list(map(fam2, imena2))
print(familii_m2)
```

Второе решение — с использованием **`lambda`**-функции и однострочного **`if`**/**`else`**:

```
familii_m3 = list(map(lambda ime: ime[:-1] + 'ев' if ime.endswith('ей') \
                      or ime.endswith('ой') else ime + 'ов', imena2))
print(familii_m2m)
```

Специальный синтаксис оператора **`if`**/**`else`** выглядит немного странно потому, что этот оператор *не производит произвольные действия, а вычисляет некоторое значение.* То есть в оператор нельзя поместить что угодно, например, там нельзя создавать локальные переменные или изменять объекты, запуская методы типа `sort` и `append`. Вместо этого можно вычислить значение в зависимости от условия. Под значением мы подразумеваем результат любого типа:

числовой, текстовый, список, кортеж, массив или что-то ещё. Сначала пишется выражение, которое будет вычислено в случае, если условие истинно, затем после ключевого слова **if** — само условие, причём оно может быть сложным, то есть разрешено использование логических операторов **not**, **and** и **or**. Условие завершается ключевым словом **else**, после которого идёт другое выражение, которое вычисляется, если условие ложно. Внимание: выражение *обязано* выдавать результат в любом случае, то есть в отличие от обычного **if** тут нельзя обойтись без ветки **else**! Если вы попробуете скормить интерпретатору что-то вроде следующей команды:

```
familii_m3 = list(map(lambda ime: ime[:-1] + 'ев' if ime.endswith('ей')))
```

то получите сообщение `SyntaxError invalid syntax` после конца строки, то есть программа даже не запустится: с точки зрения Python такая писанина — вообще не программа. В этой ситуации некоторые внимательные пользователи, следуя ранее полученным рекомендациям, начинают искать непарные скобки или пропущенные запятые, но дело не в них, хотя мы не будем обещать, что этой проблемы в вашей программе не будет: при использовании лямбд непарные скобки и неправильно расставленные запятые довольно часто встречаются и у опытных программистов на Python. Но в приведённом примере и со скобками, и с запятыми всё в порядке, дело именно в том, что в строковом условном операторе нет блока **else**.

Надо понимать, что лямбды созданы всё же для довольно простых условий. Например, вариант с тремя ветками вычислений, реализуемый стандартно через **elif**, с помощью лямбд невозможен. Правда, это ограничение можно обойти, если использовать вложенную конструкцию **if**/**else** внутри внешней такой же конструкции. Давайте рассмотрим ещё чуть более усложнённый вариант решения предыдущей задачи: добавим возможность создавать фамилии от имён первого склонения, как мужских, так и женских, что в русском языке реализуется с помощью суффикса «-ин». Из списка имён, заканчивающихся на твёрдый согласный, на «-а/-я» (например, «Илья», «Никита», «Наталья») или на «-ей/-ой» (например, «Андрей», «Сысой»), создайте список фамилий, образованных от них путём добавления суффиксов «-ов/-ев/-ин».

Решение с использованием формально однострочной лямбды.

```
familii_m3 = list(map(lambda ime: ime[:-1] + 'ев' if ime.endswith('ей') \
             or ime.endswith('ой') else ime[:-1] + 'ин' if ime.endswith('а') \
             or ime.endswith('я') else ime + 'ов', imena3))
print(familii_m3)
```

Как видим, наше формально однострочное решение плохо умещается даже в три строки (оператор \ подавляет конец строки и служит для переноса текста, если строка чересчур велика и поэтому плохо умещается на экране), поэтому пришлось немного пожертвовать стилем форматирования и сместить начала второй и третьей строк влево по сравнению с тем, как среда автоматически их размещает — попробуйте набрать текст этой программы сами и вы увидите, что

среда программирования ставит оба **or** под словом **lambda**. При этом читать такой однострочник оказывается весьма затруднительно. Тут, похоже, мы достигли предела разумного и продолжать пытаться впихнуть логику работы в рамки анонимной функции значит уподобляться герою анекдота про Perl-программистов, пишущему крутые программы однострочники, в которых способен разобраться только он сам и то только первые 15 минут после написания. Всё же код должен быть понятным: это важно и для последующей модификации (вспомним, что мы уже дважды переделывали наше решение), и для поиска ошибок, и в целях обучения/заимствования другими программистами. Мы оставляем читателя самостоятельно написать решение с использованием вместо лямбды нормальной именованной функции.

Бывает так, что функция, передаваемая первым аргументом в **map**, должна принимать два или три, или более аргументов. В этом случае разрешено передавать в **map** не два аргумента: функцию и последовательность, а три, четыре или более: функцию и несколько последовательностей. Вот как выглядит решение задачи о сложении трёх списков поэлементно в функциональном стиле:

```
list(map(lambda x, y, z: x + y + z, xlist, ylist, zlist))
```

10.2.3 Сортировка с использованием функций

В этом разделе будет рассмотрена довольно специфическая возможность Python: использование функций для сортировки. Эта возможность связана с тем, что в Python есть стандартная функция **sorted**. Мы уже рассматривали её ранее. Напомним, что она отличается от встроенного метода списков `sort` двумя качествами: во-первых, может принимать любой перечисляемый объект, а не только список, во-вторых, создаёт новый список вместо того, чтобы изменять исходный. В Python на самом деле очень мало стандартных функций, встроенных в ядро языка, большинство полезных функций либо помещены в модули, например, модуль `math`, либо представлены как методы отдельных типов данных, как популярный метод списков `append` или использованный нами ранее метод строк `endswith`. Функция **sorted** в Python создана такая, какая она есть, в том числе для применения в функциональном программировании. Дело в том, что она имеет дополнительный именованный параметр `key`, который изначально имеет значение `None`. Значение по умолчанию `None` обычно используется для реализации некоторого стандартного поведения. В случае функции **sorted** сортировка производится естественным для списка образом: числа — по значению, строки — по алфавиту, списки и кортежи — в соответствии с содержимым (сначала сравниваются первые элементы, если они равны — вторые и т. д.). Но такой способ сортировки может быть далёк от того, что нам надо. Поэтому можно написать функцию, которая будет возвращать какое-нибудь упорядоченное значение, например, числовое и использовать эти значения для сортировки. Вот пример с именами ниже.

Необходимо отсортировать исходный список имён так, чтоб сначала шли имена с окончанием на «-й», потом — на «-а/-я», а затем уже все остальные.

Для решения давайте сопоставим разным типам имён числовые значения: пусть именам на «-й» соответствует 1, именам на «-а/-я» — 2, а всем остальным — 3. Тогда решение с помощью **lambda**-функции будет выглядеть следующим образом:

```
imena4 = ['Агафон', 'Балда', 'Веденей', 'Григорий', 'Демьян',
'Евстафий', 'Ждан', 'Забава', 'Илья']
imena_sorted = sorted(imena4, key = lambda ime: 1 if ime.endswith('й') \
                      else 2 if ime[-1] in ('а', 'я') else 3)
```

Тут, чтобы не использовать метод **endswith** в комбинации с оператором **or** и сократить запись, мы использовали оператор **in**, проверяющий вхождение последнего символа строки в последовательность, в нашем случае — кортеж. Вместо лямбды можно использовать и именованные функции, если сравнение сложное и перебор большой. Обратите внимание, что при задании функции для сортировки с помощью именованного параметра **key**, другой именованный параметр функции **sorted** — **reverse** — игнорируется! То есть весь контроль за порядком теперь находится в функции.

10.2.4 Выборка из списков, функция filter

Есть задачи, когда из списка нужно выбрать только некоторые значения, подчиняющиеся определённому правилу. Для примера напишем программу, которая будет из списка имён выбирать только те, что начинаются на определённую букву.

Первое возможное решение — с помощью рекурсивной функции.

```
imena = ['Агафон', 'Богдан', 'Веденей', 'Григорий', 'Демьян', 'Евстафий',
         'Евдокия', 'Доброслава', 'Глафира', 'Варвара', 'Божена', 'Агата']
def beginA(imena, imenaA, L):
   if len(imena) > 0:
      if imena[0].startswith(L):
         return beginA(imena[1:], imenaA+[imena[0]], L)
      else:
         return beginA(imena[1:], imenaA, L)
   else:
      return imenaA

let = input ('Введите заглавную букву: ')
print(beginA(imena, [], let))
```

Второе решение — с помощью встроенной функции **filter** и лямбды:

```
print(list(filter(lambda ime: ime.startswith(let), imena)))
```

В первом решении был использован алгоритмический приём, довольно часто встречающийся в рекурсивных функциях при обработке списков. Как было несколько раз ранее, у нас есть два списка: исходный список `imena` и созданный на его основе список `imenaA`, куда мы будем заносить только имена, начинающиеся с символа, хранящегося в переменной `L`. Если каждый раз при рекурсивном вызове передавать весь исходный список целиком, понадобится также счётчик, который будет увеличиваться на 1 каждый раз и укажет, какой элемент исходного списка надо обрабатывать на очередной итерации. Этот же счётчик можно использовать, чтобы понять, достигнут ли конец исходного списка, чтобы выйти из рекурсии. Но вместо этого всего можно просто сокращать исходный список каждый раз на один текущий, только что обработанный элемент при передаче в рекурсию. Тогда наша функция всегда должна будет работать с нулевым элементом, а условием выхода из рекурсии будет нулевая длина списка. Такой подход, когда входной список постепенно «поедается» в рекурсивной функции мы ещё встретим далее.

Приведённое выше рекурсивное решение, тем не менее, выглядит довольно громоздко. Для случая, когда нужно выбрать из списка только часть элементов, придумана специальная функция **`filter`**, которая как и ранее рассмотренная функция **`map`**, относится к функциям высших порядков. Функция **`filter`** принимает в качестве первого аргумента функцию, результат которой должен быть логическим: `True` или `False`. Эта функция применяется последовательно ко всем элементам второго аргумента функции **`filter`** — итерируемого объекта, например, списка. Если при применении к очередному элементу функция выдаст `True`, значит, этот элемент включается в создаваемую последовательность, если выдаст `False` — значит, нет. То есть выходная последовательность будет состоять только из тех элементов исходной, для которых функция — первый аргумент **`filter`** — вернула `True`. Изящество второго решения состоит в том, что удалось воспользоваться встроенным методом строки `startswith`, возвращающим как раз логическое значение.

10.2.5 Свёртка списков с помощью функции `reduce`

Выше мы рассмотрели задачи генерации нового списка и преобразования одного списка в другой. Теперь рассмотрим последнюю из трёх — задачу свёртки, или редукции списка, когда в результате его обработки получается одно значение. Частными случаями редукции являются поиск минимума, максимума, суммы, длины списка и др. Для многих (самых популярных) таких операций Python имеет специализированные стандартные функции: **`min`**, **`max`**, **`sum`**, **`len`**. Поэтому при переходе с Pyhton2 на Python3 функцию **`reduce`** вынесли в модуль `functools`, хотя ранее она была в ядре языка, как **`map`** и **`filter`**. Естественно, любую функцию высших порядков можно заменить рекурсией. Для иллюстрации рассмотрим довольно простую задачу о поиске самого длинного имени в списке. Традиционно для этой главы приведём два функциональных решения: через рекурсию и через функцию высшего порядка, в данном случае — **`reduce`**.

Первое решение — через рекурсию.

```
def reductor(wort, woerter):
    if len(woerter) > 0:
        wort2 = wort if len(wort) >= len(woerter[0]) else woerter[0]
        return reductor(wort2, woerter[1:])
    else:
        return wort
print(reductor('', imena))
```

Второе решение — с помощью функции **reduce** и **lambda**-функции:

```
from functools import reduce
print(reduce(lambda a, b : b if len(b)>len(a) else a, imena))
```

В первом решении был использован уже ранее описанный приём, когда рекурсивная функция передаёт сама в себя список, сокращённый на один, обычно начальный, элемент. Однострочный оператор **if**/**else** использован для сокращения объёма кода.

Функция **reduce**, использованная во втором примере, работает так: она берёт начальное значение, а если оно напрямую не задано (это её третий аргумент), то в качестве него берётся первый элемент перечисляемого объекта (второго аргумента) и последовательно применяет свой первый аргумент (функцию, в нашем случае — безымянную) к промежуточному результату, исходно равному начальному значению, и каждому элементу перечисляемого объекта. При этом функция *обязательно* должна быть функцией двух аргументов, имеющих одинаковый тип и возвращать одно значение того же типа! Иначе всё сломается.

10.3 Конвейер, частичное применение и ленивые вычисления

Давайте рассмотрим одну очень простую на первый (и не только) взгляд задачу: напишем программу, которая будет выводить в файл квадраты целых чисел от 0 до 10.

Первое решение — через поэтапное создание списков.

```
quad = list(map(lambda v: pow(v, 2), range(10)))
quads = list(map(str, quad))
with open('квадраты1.txt', 'w') as f:
    f.write('\n'.join(quads))
```

Второе решение — создаём новую функцию для вычисления квадрата и используем конвейер функций **map**:

```
from functools import partial
sqr = partial(pow, exp=2)
with open('квадраты2.txt', 'w') as f:
    f.write('\n'.join(map(str, map(sqr, range(10)))))
```

Давайте разберём, чем может не устроить нас первый вариант. Во-первых, нам надо вычислить квадраты: это уже знакомая нам задача по преобразованию одного списка (в данном случае, диапазона **range**) в другой, которую проще всего решить с использованием функции **map** и функции возведения в квадрат. Проблема в том, что такой функции в Python нет, а есть более общая функция **pow**, возводящая что угодно в какую угодно степень, большая универсальность которой оказывается в нашем случае проблемою: нам придётся передавать два списка — для каждого из аргументов. Чтобы не делать этого, была применена лямбда, которая на месте сделала нам нужную функцию, чтобы не объявлять её через **def**. Вместо этого можно было бы всё же передать два списка, например, известным нам способом — путём умножения на 10 списка, состоящего из единственного числа 2:

```
quad = list(map(pow, range(10), 10*[2]))
```

Однако такой вариант плох тем, что мы создали ещё один дополнительный объект в памяти: помним, что **range** тем и хорош, что не хранит все значения одновременно, а выдаёт их по требованию. Раз так, можно найти аналогичный итератор, который выдавал бы нам одно и то же значение каждый раз. Такой в стандартной библиотеке Python есть, называется он **repeat**, принимает два аргумента: что итерировать и сколько раз и лежит в модуле **itertools**:

```
from itertools import repeat
quad = list(map(pow, range(10), repeat(2, 10)))
```

Второе решение предлагает ещё один способ реализации функции возведения в квадрат, которое называют *частичным применением*: мы создаём новую функцию на основе предыдущей путём фиксирования части (в нашем случае одного — второго) аргументов. Частичное применение в Python реализовано с помощью функции **partial**, которая не стандартная, а должна быть импортирована из стандартного модуля **functools**, откуда ранее мы брали **reduce**. Эта функция принимает на вход в качестве первого аргумента уже существующую функцию, далее идёт перечисление значений её аргументов, которые нужно зафиксировать (лучше всего, конечно, обращаться к аргументам по имени, если это возможно, потому что так сразу понятно, что фиксируется), а выдаёт **partial** тоже функцию, но с уже урезанными возможностями. В общем, очень функционально! Фактически же **partial** предлагает уже третий наряду с **def** и **lambda** механизм объявления функции и, как правило, одну и ту же проблему можно решить любым из этих трёх способов.

Теперь сообразим, что сам по себе список **quad** нам не нужен, он является промежуточным звеном вычислений, его сразу нужно подать на вход следующей функции **map**, ведь писать в файл нужно строки, а не числа. Такой подход называется *конвейером*: результат выполнения одной функции высшего порядка мы сразу же подаём на вход другой. В императивном программировании это может соответствовать двум циклам, идущим один за другим, либо двум операциям внутри одного цикла.

В действительности, функция **map** при своём объявлении ещё ничего не вычисляет, как и ряд других функций, в том числе **filter**, **zip**, **enumerate**, **range**, а также большинство функций из модулей `functools` и `itertools`. Вычисление происходит в момент, когда требуется преобразовать результат в какой-то конечный объект. Например, если вы явно преобразуете результат к списку с помощью **list** или кортежу с помощью **tuple**. В случае со вторым решением вычисления происходят в момент вызова метода `join`, формирующего строку на выходе. То есть в конце любого конвейера обязательно стоит какая-то функция, приводящая все вычисления к конечному объекту, находящемуся в памяти, потому что промежуточные результаты функций типа **map** в памяти целиком не находятся! Такие вычисления называются *ленивыми* и имеют два основных преимущества:

1. часть вычислений, результаты которых оказываются не нужны впоследствии, не производятся,

2. результаты других вычислений не хранятся в памяти постоянно, не создаются промежуточные структуры данных, вместо этого они сразу передаются вперёд по конвейеру.

Ленивые вычисления в целом характерны для функциональных языков потому, что функциональные языки тяготеют к формированию конвейера и создают новые объекты в памяти каждый раз вместо того, чтобы изменять уже существующие. В императивных языках аналогичные действия выполняются с помощью циклов, которые работают с объектами непосредственно в памяти, в том числе с изменяемыми, поэтому большого смысла в поддержке ленивых вычислений там нет.

В конце раздела рассмотрим пример задачи, при решении которого будут применены почти все ранее обсуждавшиеся приёмы функционального программирования, кроме рекурсии.

Пример задачи 34 (ЕГЭ) Есть файл с данными в четыре столбца: фамилия, имя, отчество абитуриентов и их балл ЕГЭ (целое число от 0 до 100). Напишите программу, которая запишет в новый файл только тех из них, чья оценка «хорошо» или «отлично», если оценки «удовлетворительно» начинаются от 28 баллов, «хорошо» — от 48 баллов, а «отлично» — от 66 баллов. В файле могут быть пустые строки.

Решение задачи 34 Первое решение — через перебор.

```
def abr(ime):
    return ime[0]+'.'
def unrec(record):
    fam, ime, otch, ocen = record.split()
    oceni = int(ocen)
    fio = ' '.join((fam, abr(ime), abr(otch)))
    if oceni < 28:
```

```
        return fio + ' неудовлетворительно'
    elif oceni < 48:
        return fio + ' удовлетворительно'
    elif oceni < 66:
        return fio + ' хорошо'
    else:
        return fio + ' отлично'
with open('Оценки.txt', 'r', encoding='utf-8') as f1:
    with open('хорошие_оценки.txt', 'w') as f2:
        f2.write('\n'.join(filter(lambda v: v.endswith('хорошо') or v.endswith('отлично')
                          map(unrec, filter(lambda v: v.strip(), f1.readlines())))))
```

Второе решение отличается от первого только модификацией функции `unrec` с использованием словаря и **`reduce`**:

```
def unrec(record):
    fam, ime, otch, ocen = record.split()
    oceni = int(ocen)
    minball = [(0, 'неудовлетворительно'), (28, 'удовлетворительно'),
               (48, 'хорошо'), (66, 'отлично')]
    return ' '.join((fam, abr(ime), abr(otch),
                     reduce(lambda u, v: u if v[0] > oceni else v, minball)[1]))
```

Первое решение обзавелось двумя функциями: одна — `abr` — совсем простенькая и преобразует имя или отчество к инициалу с точкою. Вторая, использующая первую, обрабатывает запись одного ученика и перебирает все варианты оценок, выдавая фамилию с инициалами и оценку вместо балла ЕГЭ. Далее мы открываем исходный файл на чтение, выходной файл — на запись, читаем из исходного файла все записи в список и используем конвейер из функций высшего порядка. Сначала внутренняя функция **`filter`** пропускает дальше только непустые строки, то есть те, где есть хотя бы один непробельный символ; чтобы уменьшить писанину, мы пользуемся тем, что пустая строка интерпретируется в Python как ложь, а непустая — истина. Затем функция **`map`** обрабатывает все извлечённые непустые, то есть прошедшие через фильтр, строки с помощью функции `unrec`, передавая их далее второму фильтру, который пропускает только строки, заканчивающиеся на «хорошо» или «отлично». Получившийся набор строк (списком формально он не является) попадает в метод `join`. Поскольку `join` — фактически единственный метод, создающий объект в памяти, реальное исполнение всех других функций происходит именно в нём, до этого все вычисления — ленивые. Сложно сказать, какая реализация функции `unrec` однозначно лучше. Первая в целом нагляднее, вторая короче и надёжнее, потому что, например, при увеличении числа возможных оценок она гораздо проще правится и перепутать знак неравенства в ней можно только один раз, причём результат будет сразу заметен.

В первом решении есть перебор вариантов. В Python перебор чаще всего реализуется либо с помощью набора **`elif`**, как это и сделано, либо с помощью обращения к словарю по ключу. Но в нашем случае перебор имеет место не

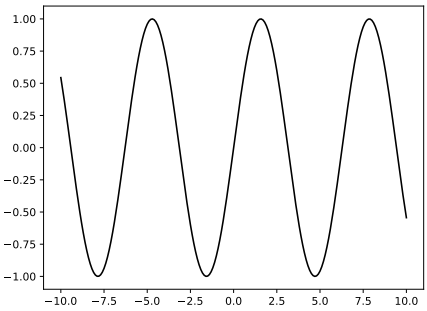


Рис. 10.1. Иллюстрация к задаче 35.

между значениями, а между диапазонами, поэтому прямой подход со словарём неприменим. Тем не менее, его можно реализовать, если сделать список значений минимального балла и соответствующей ему оценки и использовать функцию **reduce**. Кстати, встроенная в **lambda**-функция обращается на чтение к переменной oceni, определённой в родительской функции, то есть имеет место замыкание!

10.4 Примеры решения заданий

Пример задачи 35 Используя модуль math без использования модуля numpy, протабулируйте функцию $\sin(t)$ на отрезке $t \in [-10; 10]$ и постройте график средствами модуля matplotlib.

Решение задачи 35

```
import math
import matplotlib.pyplot as plt
dt = 0.1 #шаг по времени
time = list(map(lambda x: x*dt, range(-100, 101))) #создаём временной отрезок
f = list(map(math.sin, time)) #вычисляем значения синуса
plt.plot(time, f, color = 'black')
plt.show()
```

Пример задачи 36 С помощью функции **map**, других встроенных функций: **sum**, **max**, **min**, **pow**, **len** и при необходимости **lambda**-функций для списка, состоящего из трёх списков чисел разной длины, выдайте список среднеквадратичных отклонений.

Решение задачи 36

```
from functools import partial
def std(L0):
    '''функция, возвращающая среднеквадратичное отклонение для одного списка'''
    mu = sum(L0)/len(L0) #вычисляем матожидание
    sqr = partial(pow, exp=2) #создаём функцию, возводящую только в квадрат
    return (sum(map(sqr, map(lambda x: x-mu, L0)))/len(L0))**0.5

L = [[-3, 4, 8, -1, -10], [0, 2, 2, 0], [-1, 4, -2, 1, -1, 0, 3, 2, 3]]
print(list(map(std, L)))
```

Вывод программы:

```
[6.151422599691879, 1.0, 2.0]
```

Пример задачи 37 С помощью функции **sorted** с дополнительным аргументом отсортируйте список слов, полученных делением строки документации функции округления **round()**, по длине: от самого короткого к самому длинному.

Решение задачи 37

```
sorted(round.__doc__.split(), key = lambda v: len(v))
```

Вывод программы:

```
['a', 'a', 'to', 'in', 'is', 'an', 'if', 'is', 'or', 'as', 'be', 'The', 'the',
 'has', 'the', 'the', 'may', 'same', 'type', 'Round', 'given', 'value', 'None.',
 'value', 'number', 'return', 'return', 'decimal', 'digits.', 'integer',
 'ndigits', 'omitted', 'number.', 'ndigits', 'precision', 'Otherwise',
 'negative.']
```

Пример задачи 38 Используя функцию **map** и другие методы функционального программирования, вычислите средний квадрат чисел в файле, числа расположены в столбик, результат выведите на экран.

Решение задачи 38

```
f = open('Числа.txt', 'r')
L = list(map(int, f.readlines()))
print(sum(map(lambda x: x**2, L))/len(L))
```

Пример задачи 39 Используя функции **map**, **filter** и другие методы функционального программирования, обработайте текст, лежащий в файле (возьмите у преподавателя), найдите все слова, начинающееся на любую букву, введённую пользователем (не важно, заглавная или строчная буква), слова разделены пробелом. Результат выведите на экран. Все знаки препинания перечислены в переменной `punctuation`, определённой в стандартном модуле `string`. Чтобы сделать любую букву строчной можно использовать стандартный метод `lower`.

Решение задачи 39

```
from string import punctuation
def del_pun(s):
    '''удаляет знаки пунктуации в начале и конце строки'''
    if len(s)==0:
        return s
    else:
        if s[-1] in punctuation:
            return del_pun(s[:-1])
        else:
            if s[0]=='"': #вначале могут быть только кавычки
                return s[1:]
            else:
                return s
let = input('Введите русскую букву: ').lower() #делает букву строчной
f = open('Текст.txt', 'r', encoding='utf-8')
S = f.read().split() #разделяем строку на подстроки, разделённые пробелами
print(list(filter(lambda l: l.lower().startswith(let), map(del_pun, S))))
```

Вывод программы:

```
Введите русскую букву: З
['заросшей', 'задавленные', 'знаем', 'заключению', 'зла']
```

10.5 Задания на применение функционального программирования

Задание 38 Выполнять три задания в зависимости от номера в списке. Необходимо сделать задания № m, № $m+5$, № $m+10$, где $m = (n-1)\%5+1$, n — номер студента в списке группы в алфавитном порядке.

Используя модуль `math` без использования модуля `numpy`, протабулируйте следующие функции и постройте графики средствами модуля `matplotlib`.

1. $x^2 + 10x + 1$ на отрезке $x \in [-2; 2]$;

2. $x^3 - 4$ на отрезке $x \in [-2; 2]$;
3. $(x-2)^4$ на отрезке $x \in [0; 2]$;
4. $\cos(2\pi t)$ на отрезке $t \in [-10; 10]$;
5. $\frac{1}{t}\cos(2\pi t)$ на отрезке $t \in [1; 10]$;
6. $e^{-t}\cos(2\pi t)$ на отрезке $t \in [-10; 10]$;
7. $4\sin(\pi t + \pi/8) - 1$ на отрезке $t \in [-10; 10]$;
8. $2\cos(t-2) + \sin(2t-4)$ на отрезке $t \in [-20\pi; 10\pi]$;
9. $\ln(x+1)$ на отрезке $x \in [0; e-1]$;
10. $\log_2(|x|)$ на отрезке $x \in [-4; 4]$ за исключением точки $x = 0$;
11. 2^x на отрезке $x \in [-2; 2]$;
12. e^x на отрезке $x \in [-2; 2]$;
13. 2^{-x} на отрезке $x \in [-2; 2]$;
14. $\sqrt[3]{x}$ на отрезке $x \in [1; 125]$;
15. $\sqrt[5]{x}$ на отрезке $x \in [1; 32]$.

Задание 39 Выполнять одно задание с номером $(n-1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

С помощью функции **`map`**, других встроенных функций: **`sum`**, **`max`**, **`min`**, **`pow`**, **`len`** и при необходимости **`lambda`**-функций для списка, состоящего из нескольких списков чисел разной длины, выдайте список:

1. сумм;
2. максимумов;
3. минимумов;
4. средних значений;
5. произведений.

Задание 40 Выполнять одно задание с номером $(n-1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

С помощью функции **`sorted`** с дополнительным аргументом отсортируйте значения в списке по следующему правилу (можно пользоваться как лямбдами, так и обычными именованными функциями по вашему выбору):

1. Список слов получите делением строки документации функции `sorted()`. Отсортируйте полученные слова по длине: от самого длинного к самому короткому.

2. Список слов получите делением строки документации функции `sorted()`. Отсортируйте полученные слова по длине: от самого короткого к самому длинному.

3. Список слов получите делением строки документации функции `help()`. Отсортируйте полученные слова по длине: от самого длинного к самому короткому.

4. Список слов получите делением строки документации функции `help()`. Отсортируйте полученные слова по длине: от самого короткого к самому длинному.

5. Список слов получите делением строки документации функции `pow()`. Отсортируйте полученные слова по длине: от самого длинного к самому короткому.

6. Список слов получите делением строки документации функции `pow()`. Отсортируйте полученные слова по длине: от самого короткого к самому длинному.

Задание 41 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Используя функцию `map` и другие методы функционального программирования, обработайте список чисел, лежащих в файле, результат выведите на экран.

1. Вычислите сумму квадратов чисел в файле, числа расположены в столбик.

2. Вычислите сумму кубов чисел в файле, числа расположены в столбик.

3. Вычислите сумму модулей чисел в файле, числа расположены в столбик.

4. Вычислите средний квадрат чисел в файле, числа расположены в столбик.

5. Вычислите средний модуль чисел в файле, числа расположены в столбик.

Задание 42 Выполнять три задания в зависимости от номера в списке. Необходимо сделать задания № m, № $m+5$, № $m+10$, где $m=(n-1)\%5+1$, n — номер студента в списке группы в алфавитном порядке.

Используя функции `map`, `filter`, `reduce` и другие методы функционального программирования, обработайте текст, лежащий в файле (возьмите у преподавателя), результат выведите на экран.

1. Найдите самое длинное слово, начинающееся на «у» (не важно, заглавная или строчная буква).
2. Найдите самое короткое слово, начинающееся на «г» (не важно, заглавная или строчная буква).
3. Найдите самое длинное слово, начинающееся с заглавной буквы «Н».
4. Найдите самое короткое слово, начинающееся с заглавной буквы «В».
5. Найдите самое короткое слово, начинающееся со строчной буквы «д».
6. Найдите все слова с окончанием на «-на», начинающиеся с заглавной буквы.
7. Найдите самое короткое слово с окончанием «-ся».
8. Найдите самое длинное слово с окончанием «-ся».
9. Найдите самое короткое слово с окончанием «-ый».
10. Найдите самое длинное слово с окончанием «-ый».
11. Найдите строку, в которой больше всего слов, слова из одной–двух букв не считаются.
12. Найдите строку, в которой меньше всего слов, слова из одной–двух букв не считаются.
13. Найдите строку, в которой больше всего слов, начинающихся с заглавной буквы.
14. Найдите строку, в которой меньше всего слов, начинающихся с заглавной буквы.
15. Найдите самую длинную по числу символов строку.

Глава 11

Графический интерфейс пользователя средствами модуля `tkinter` и объектно-ориентированное программирование

В настоящее время трудно представить себе программу для персонального компьютера или ноутбука, для которой не было бы графического интерфейса пользователя с менюшками, кнопочками, всплывающими диалоговыми окнами, полями ввода и прочими аттрибутами. Трудно, но на самом деле у вас на компьютере таких программ запущено несколько десятков, просто вы их не видите, потому что запускают их либо другие программы, либо сама система. Тем не менее, следует признать, что графический интерфейс — необходимая часть многих приложений и программировать его вам вполне возможно придётся.

В старые времена, которые даже авторы этих строк застали лишь краем глаза, для программирования графического интерфейса необходимо было изучить возможности, предоставляемые непосредственно операционною системою, будь то Win API для Windows или команды оболочки X11 для UNIX-подобных систем. Однако уже достаточно давно, примерно с 1998–1999 года все основные программы пишутся с использованием специальных библиотек виджетов — графических примитивов вроде кнопок, полей ввода, статических списков и прочих. Конечно, практически все библиотеки имеют слои совместимости с Python, хотя ни одна из них не была написана для программирования на этом языке изначально. Более того, существует множество популярных пользовательских программ, написанных на Python, как и графических интерфейсов к консольным утилитам UNIX.

Наиболее распространённая в настоящее время библиотека — Qt, причём одновременно используются несколько принципиально разных поколений: 5-ое, 4-ое, а кое-где и 3-е, внутри каждого из которых создано множество версий. Qt

очень мощная и продвинутая библиотека, созданная весьма давно и целиком написанная на C++. Изначально она поставлялась под коммерческою лицензией, но в настоящее время доступна свободно. Для Python существует несколько альтернативных реализаций слоёв совместимости с нею, причём в разное время то одни, то другие объявлялись приоритетными: PyQt4, PySide, PyQt5. Поскольку Qt — это всё-таки большая и очень мощная библиотека, её изучение сопряжено со значительными усилиями, а нормальная работа с нею невозможна без знания C++ и внутренней её организации (а во многом ещё и знания специального языка разметки QML).

Другая основная свободная библиотека виджетов — GTK+. Современная её версия — 4-я, большинство программ используют 3-ю и всё ещё много собранных с использованием GTK+ 2-ой версии. Проект PyGTK, предоставлявший возможности использования GTK2 для Python, завершил активное развитие в 2011 году, хотя использование PyGTK с Python версий 2.x возможно и поныне. К сожалению, нормальный порт GTK под Windows отсутствует, а существующие имеют много ограничений. Во времена GTK2 эту проблему решала весьма популярная библиотека wxWidgets (первоначальное название — wxWindows), использовавшая под Linux и Mac GTK+, а под Windows обращавшаяся непосредственно к Win API. Низкая популярность GTK версии 3 и медленное развитие проектов wxWidgets и PyGTK привели к тому, что в настоящее время использование GTK для написания новых программ на Python стало непопулярно. Использование PyGObject — наследника PyGTK — для Python 3 вполне возможно и в Интернете даже можно найти довольно подробные руководства, правда, почти только на английском.

Существуют ещё несколько относительно распространённых библиотек, в частности fltk и Tk, интерфейс к которой, называемый tkinter, будет рассмотрен далее, поскольку tkinter — это стандартная библиотека виджетов для языка Python, поставляемая вместе с интерпретатором. Первоначально библиотека Tk была написана для языка tcl (его по-русски часто называют «тикль»), поэтому его реализация в Python несёт отпечаток некоторых особенностей tcl. Благодаря простоте, малому размеру кода и хорошей переносимости, а также большой стабильности tcl/Tk (изменения вносятся редко и, как правило, не влияют на поведение ранее написанного кода), автор Python — Гвидо ван Россум принял решение использовать Tk в качестве стандартного графического интерфейса, причём это решение не поменялось и с появлением более продвинутых средств, таких как Qt или GTK+.

Изучение библиотеки tkinter будем излагать на примере создания собственных нескольких простых офисных и инженерных программ.

11.1 Калькулятор

Рассказ о графических интерфейсах пользователя начнём с написания простого калькулятора. Конечно, у вас в системе калькулятор уже есть: это несложное приложение, основными элементами которого являются строка, куда можно

вводить цифры (обычно в верхней части), и набор кнопок. Но тем интереснее начать с этого примера, поскольку скоро вы увидите, что легко сможете написать такую программу сами.

При написании программ с графическим интерфейсом, как правило, предлагают начинать с написания собственного **класса** на основе стандартного класса приложения. Библиотека tkinter также предлагает такую возможность, но мы ею пользоваться пока не будем, потому что для простых приложений без этого легко обойтись, так что смысл и польза от написания такого класса, наследования и переопределения функций будут неочевидны.

Итак, нам нужны поле ввода и кнопка, много кнопок: для цифр, знака разделителя целой и дробной частей (точка или запятая), знаки арифметических операций, а также кнопки для смены знака и для удаления последнего неверно введённого символа. Но сначала нам нужно создать главное окно-форму, на которой текстовое поле и кнопки будут размещаться. Для этого необходимо импортировать модуль tkinter и его подмодуль `ttk`, в котором хранятся более новые и красивые варианты виджетов, и создать **форму** (назовём её `prog`), на которую будут помещаться прочие виджеты:

```
import tkinter
from tkinter import ttk
prog = tkinter.Tk()
```

Теперь создадим **текстовое поле**:

```
stext = tkinter.StringVar(value='')
widtext = ttk.Entry(textvariable=stext)
```

Разместим его на форме и запустим главный цикл обработки событий командой `prog.mainloop()`:

```
widtext.grid(column=0, row=0)
prog.mainloop()
```

В итоге получим вот такую форму с текстовым полем на ней (рис. 11.1).

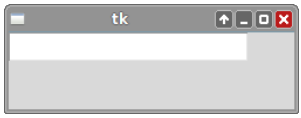


Рис. 11.1. Форма `Tk` и текстовое поле `ttk.Entry`.

Само текстовое поле создаётся как объект — экземпляр класса `Entry`, размещённого в модуле `ttk`, мы назвали наше поле `widtext`. В принципе, можно при создании `widtext` вообще не задавать никаких аргументов, но нам нужно будет как-то читать информацию из поля, чтобы можно было производить вычисления, и как-то менять его содержимое при нажатии кнопок. Поэтому мы создали

специальную переменную типа StringVar (хранится в модуле tkinter) с именем stext и первоначальным значением '' (пустая строка) и связали её с нашим текстовым полем с помощью именованного аргумента textvariable. Такой аргумент есть у нескольких виджетов, не только у текстового поля. Представленный способ может показаться несколько усложнённым, кажется, что вместо переменной типа tkinter.StringVar можно было бы использовать простую текстовую переменную, но это не так: нам нужно, чтобы такая переменная обновлялась автоматически, когда пользователь меняет содержимое поля (вводит туда символы). Простая переменная типа **str** такого функционала не имеет.

Созданное поле не появится на форме до тех пор, пока мы не разместим его там. В данном случае это сделано методом grid, который есть у всех основных виджетов. Метод принимает несколько параметров, самые востребованные из которых column и row — номер колонки и ряда, в котором размещён виджет. При размещении виждетов на форме с помощью grid считается, что каждый из них размещается на сетке — как фигуры на шахматной доске.

Создадим теперь **кнопку**, которая позволит нам вводить какое-нибудь число, например, «1»:

```
btn1 = ttk.Button(text='1')
btn1.bind('<Button-1>', renov1)
btn1.grid(column=0, row=1)
```

Первая строчка создаёт кнопку с именем btn1, на которой будет написано «1» — это достигается с помощью аргумента text='1'. Третья строка размещает кнопку на форме в той же колонке, что и текстовое поле, но в следующей строке (под полем ввода снизу). Основной интерес представляет для нас команда btn1.bind('<Button-1>', renov1), связывающая нажатие на кнопке левой клавишей мыши (это определяет '<Button-1>') с функцией renov1, которую мы ещё не написали. Функция будет вызываться при каждом нажатии.

Описать функцию нужно до вызова метода bind, лучше всего в начале программы, сразу после импорта модулей:

```
def renov1(event):
    stext.set(stext.get() + '1')
```

Функция обращается к методу get() переменной stext, связанной с нашим текстовым полем. Этот метод переводит внутреннее представление содержимого этой переменной в стандартную строку. Обратная ей функция **set()** помещает стандартную строку Python в переменную типа StringVar. Аргумент этой функции event не имеет для нас практического значения, поскольку вызывать её сами мы никогда не будем. Функция будет вызываться только тогда, когда нажимается кнопка, и передаваться этот аргумент будет автоматически. Параметр event до некоторой степени «магический», хотя никакой магии там конечно же нет: просто часть реализации скрыта от нас не только внутри Python, но и внутри интерпретатора tcl.

Важно! Строка с `prog.mainloop()` должна оказаться в самом конце программы по аналогии со строкой `plt.show()` при построении графиков. Теперь мы получим форму с текстовым полем и кнопкой (рис. 11.2).

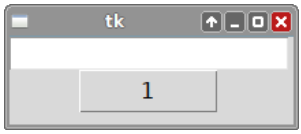


Рис. 11.2. Форма `Tk`, текстовое поле `ttk.Entry` и кнопка `ttk.Button`.

Итак, у нас есть программа, в которой есть кнопка, по нажатию которой в поле ввода появляется дополнительная единица. Дальше наделаем ещё кнопок для цифр (многоточием мы заменили ещё 6 кнопок, которые делаются аналогично):

```
# Кнопка "1":
btn1 = ttk.Button(text='1')
btn1.bind('<Button-1>', renov1)
btn1.grid(column=0, row=1)
# Кнопка "2":
btn2 = ttk.Button(text='2')
btn2.bind('<Button-1>', renov2)
btn2.grid(column=1, row=1)
...
# Кнопка "9":
btn9 = ttk.Button(text='9')
btn9.bind('<Button-1>', renov9)
btn9.grid(column=2, row=3)
# Кнопка "0":
btn0 = ttk.Button(text='0')
btn0.bind('<Button-1>', renov0)
btn0.grid(column=1, row=4)
```

Для каждой из кнопок придётся написать свою функцию:

```
def renov1(event):
    stext.set(stext.get() + '1')
def renov2(event):
    stext.set(stext.get() + '2')
...
def renov9(event):
    stext.set(stext.get() + '9')
def renov0(event):
    stext.set(stext.get() + '0')
```

Чтобы получившаяся программа выглядела лучше, заставим наше текстовое поле занимать сразу три колонки (параметр `columnspan=3` в методе `grid`)

и зададим длину в символах (иначе поле будет коротким и станет смотреться некрасиво), добавив параметр `width=30` при создании поля ввода:

```
widtext = ttk.Entry(textvariable=stext, width=30)
widtext.grid(column=0, row=0, columnspan=3)
```

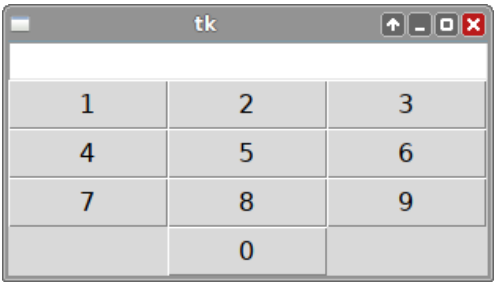


Рис. 11.3. Недоделанная программа–калькулятор с кнопками для всех цифр.

Выглядит сносно, но пока наша программа ничего не считает. К тому же для реализации весьма примитивного функционала мы написали уже около 70 строк кода, в основном — методом копирования, что в программировании крайне не приветствуется, поскольку почти всегда ведёт к ошибкам, вызванным тем, что что-то забыли исправить либо во время копирования, либо после, когда меняли и улучшали программу. Тем не менее, пока не будем обращать на это внимания и сделаем программу работающею.

Давайте для начала добавим две нужные кнопки. Кнопку «.», означающую десятичный разделитель:

```
btnP = ttk.Button(text='.')
btnP.bind('<Button-1>', renovP)
btnP.grid(column=2, row=4)
```

и кнопку удаления последнего введённого символа:

```
btnB = ttk.Button(text='<-')
btnB.bind('<Button-1>', renovB)
btnB.grid(column=3, row=1)
```

Для них также придётся написать функции, названные нами `renovP` и `renovB`, соответственно:

```
def renovP(event):
    stext.set(stext.get() + '.')
def renovB(event):
    stext.set(stext.get()[:-1])
```

Далее добавим кнопку смены знака:

```
btn_pm = ttk.Button(text='-/+')
btn_pm.bind('<Button-1>', renov_pm)
btn_pm.grid(column=0, row=4)
```

для которой пропишем функцию:

```
def renov_pm(event):
    s = stext.get()
    if s.startswith('-'):
        s = s[1:]
    else:
        s = '-' + s
    stext.set(s)
```

Эта функция определяет, начинается ли строка со знака минуса, используя метод `startswith` и если да, то отрезает его, а если нет, то наоборот — добавляет. Как обычно, приходится сначала вытащить текст из переменной типа `StringVar` методом `get`, а затем после всех операций поместить её обратно методом **`set`**.

Теперь перейдём к написанию самого сложного — функций для работы с кнопками арифметических операций. Ограничимся пока кнопками для сложения, вычитания и кнопкою «равно», которая выдаёт результат вычислений. Разместим эти кнопки:

```
btn_plus = ttk.Button(text='+')
btn_plus.bind('<Button-1>', oper_plus)
btn_plus.grid(column=3, row=2)
btn_minus = ttk.Button(text='-')
btn_minus.bind('<Button-1>', oper_minus)
btn_minus.grid(column=3, row=3)
btn_ravno = ttk.Button(text='=')
btn_ravno.bind('<Button-1>', oper_ravno)
btn_ravno.grid(column=3, row=4)
```

Теперь нужно вспомнить, как работает калькулятор. Первоначально в поле ничего нет (или стоит 0), затем вы набираете первое число и нажимаете знак арифметической операции, после чего строка ввода очищается, а само набранное ранее число заносится в память. Далее набирается новое число и нажимается новый знак. Если он — «равно», то производится ранее набранная операция и результат выводится. Если же это опять «плюс» или «минус», то ранее набранная операция всё равно производится, но результат никуда не выводится, а помещается обратно в память, и поле ввода снова очищается.

Таким образом, нужно сделать переменную, реализующую память калькулятора, скажем, в виде числа, присвоив ей начальное значение 0. Кроме того, нужно запомнить, например, в виде символа, предыдущую операцию, ведь при нажатии кнопки выполняться будет именно она. Обе эти переменные придётся сделать глобальными.

```
# Память:
hidden = 0
# Предыдущая операция:
preoper = ''
```

Тогда операции сложения, вычитания и вычисления результата («равно») можно реализовать следующим образом:

```
def oper_plus(event):
    if not hidden:
        hidden = float(stext.get())
    else:
        if preoper == '+':
            hidden += float(stext.get())
        elif preoper == '-':
            hidden -= float(stext.get())
    stext.set('')
    preoper = '+'
def oper_minus(event):
    if not hidden:
        hidden = float(stext.get())
    else:
        if preoper == '+':
            hidden += float(stext.get())
        elif preoper == '-':
            hidden -= float(stext.get())
    stext.set('')
    preoper = '-'
def oper_ravno(event):
    if hidden:
        if preoper == '+':
            hidden += float(stext.get())
        elif preoper == '-':
            hidden -= float(stext.get())
    stext.set(str(hidden))
    preoper = ''
```

Полученная программа будет выглядеть примерно как на рис. 11.4. Но только наша программа корректно работать не будет. При нажатии знака «плюс» (и любой другой операции тоже) в командном интерпретаторе, куда раньше мы выводили результаты, который запускается также и для программ с графическим интерфейсом, можно будет прочитать сообщение об ошибке:

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/usr/lib/python3.5/tkinter/__init__.py", line 1562, in __call__
    return self.func(*args)
  File "/home/lab41/kalkulilo_C.py", line 38, in oper_plus
    if not hidden:
```

```
UnboundLocalError: local variable 'hidden' referenced before assignment
```

Сообщение достаточно красноречивое: если даже вы плохо понимаете по-английски, любой интернет-переводчик даст вам возможность понять, что не так: интерпретатор считает нашу переменную `hidden`, которая реализует память, локальной в функции, причём сбой происходит в первой же строке, где она используется: `if not hidden`. Проблема заключается в том, что Python позволяет пользоваться глобальными переменными свободно далеко не всегда. По умолчанию, он как раз пытается создать локальные переменные с тем же именем. Чтобы заставить интерпретатор использовать `hidden`, а заодно и `preoper`, нужно явно объявить их глобальными, использовав директиву `global`, как это показано ниже:

```
def oper_plus(event):
    global hidden, preoper
    if not hidden:
        hidden = float(stext.get())
    else:
        if preoper == '+':
            hidden += float(stext.get())
        elif preoper == '-':
            hidden -= float(stext.get())
    stext.set('')
    preoper = '+'
```

То есть мы пришли к ситуации, когда неявная типизация, сэкономившая нам много сил до сих пор, стала проблемой: интерпретатор Python не может понять, что нужно взять уже существующую глобальную переменную, а вместо этого считает, что это локальная переменная функции, и попадает в ловушку, так как оказывается неспособен прочитать её значение до её создания. Интересно, что выводя сообщение об ошибке, интерпретатор не завершает программу аварийно.

Если поправить код трёх написанных функций для работы с операциями, наша программа, наконец, почти заработает: она сможет производить правильные вычисления при условии, что вы не ошиблись со вводом. Обычно на калькуляторе есть кнопка сброса всех предыдущих операций, давайте реализуем и её:

```
def cancel(event):
    global hidden, preoper
    preoper = ''
    hidden = 0
    stext.set('')
btnC = ttk.Button(text='C')
btnC.bind('<Button-1>', cancel)
btnC.grid(column=3, row=0)
```

11.2 Метки, флаги, радиокнопки и диалоги

Графический интерфейс не ограничивается только кнопками и полями ввода, это каждый видел, работая со многими программами. В этом разделе постараемся описать ещё несколько наиболее популярных виджетов на примере их использования.

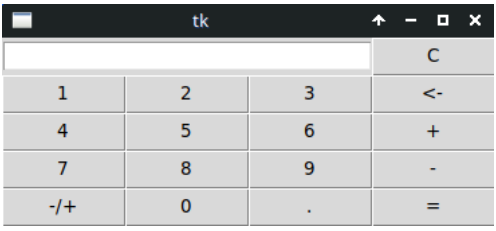


Рис. 11.4. Программа–калькулятор — первая работоспособная версия с кнопками сложения, вычитания и получения результата.

Давайте создадим программу, которая позволит заполнить файл с фамилиями учеников. Для этого создадим поле ввода и кнопку для записи в файл:

```
from tkinter import *
from tkinter import ttk
prog = ttk.Frame()
strF = StringVar()
editF = ttk.Entry(textvariable=strF)
editF.grid(column=0, row=1)
bts = ttk.Button(text='Записать')
bts.bind('<Button-1>', swrite)
bts.grid(column=0, row=5)
prog.mainloop()
```

Мы намеренно разместили поле и кнопку в 1-ой и 4-ой строках, чтобы показать, что если оставить часть строк или столбцов незаполненными (нет ни одного виджета, размещённых там), они просто будут проигнорированы и «схлопнутся». Зато это может быть полезно в будущем, если мы решим разместить дополнительные виджеты до или между уже существующими, так как в таком случае не придётся переделывать уже существующий код.

Теперь нужно сделать функцию `swrite` для записи фамилии в файл. Идея состоит в том, что при каждом нажатии кнопки мы можем дозаписывать фамилию в любой уже существующий текстовый файл или создать новый. Для выбора файла используем стандартный диалог, имеющийся в `tkinter`, для чего нужно импортировать модуль `filedialog`. В этом модуле есть функция `asksaveasfilename`, выполняющая вызов диалога сохранения в файл, сам диалог при этом писать не нужно, он уже есть в модуле и сформируется автоматически со всеми своими кнопками и подписями. Эта функция возвращает имя файла. Для дозаписи нам нужно самим открыть файл в режиме `'a'`. Получится вот что:

```
from tkinter import filedialog as fd
def swrite(event):
    fname = fd.asksaveasfilename()
    f = open(fname, 'a')
    f.write(strF.get()+'\n')
    f.close()
```

Если скомпоновать этот фрагмент с ранее написанным (импорт модуля следует перенести вверх), получится вполне работоспособная программа. Заметьте, что диалог является как бы скрытым виджетом: в отличие от всех других, рассмотренных до сих пор, он не размещается нами на форме и не виден до тех пор, пока не будет вызван по кнопке. Более того, мы вообще не создавали никакую переменную, ответственную за него, хотя это и можно сделать. В итоге (рис. 11.5) в файл с названием ‘Student list.txt’, которое мы сами вписали в диалоговом окне, будет записана одна фамилия ‘Александров’. Затем можно в поле ввода ввести следующую фамилию, нажать кнопку ‘Записать’, и в файл дозапишется вторая фамилия.

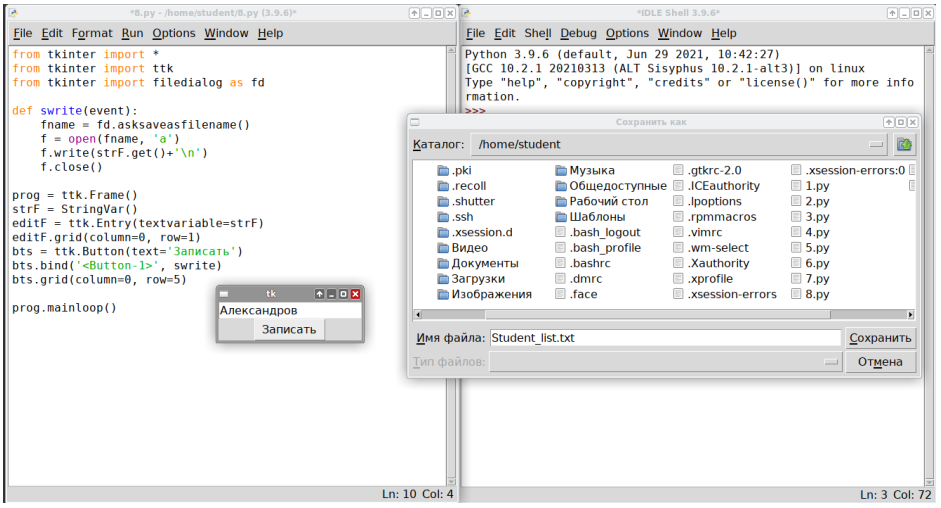


Рис. 11.5. Поле ввода `ttk.Entry`, кнопка для записи в файл `ttk.Button` и диалоговое окно `filedialog`.

Кроме функции `asksaveasfilename` есть ещё аналогичная функция `askopenfile`, вызывающая диалог чтения файла. Заметим, что обе функции в действительности только возвращают имя файла и ничего не открывают. У них есть аналоги `asksaveasfile` и `askopenfile`, которые упрощают работу, возвращая уже открытый файл, а не его имя (не нужно самим вызывать стандартную функцию `open`). Функция `asksaveasfile` открывает файл в режиме `'w'`, а функция `askopenfile` — в режиме `'r'`, поэтому при открытии файла на дозапись воспользоваться ими нельзя.

Поле ввода может изменяться пользователем, но часто на форму необходимо поместить поясняющий текст, который является статичным или может меняться только изнутри программы. Для это существует отдельный виджет «метка» `Label`. В приведённом выше примере можно добавить на форму пояснение, что вводимое слово — фамилия, не меняя код, используя зарезервированный ранее нулевой ряд.

```
labelF = ttk.Label(text='Фамилия')
labelF.grid(column=0, row=0)
```

Давайте теперь добавим ещё и возможность отметить, является ли записываемый человек взрослым или ребёнком. В этом случае дополнительное поле ввода ни к чему, так как есть всего 2 варианта. Вместо этого можно использовать специальный виджет «флаг», «флаговая кнопка» или «галочка» `CheckButton`, который мы расположим в зарезервированном третьем ряде:

```
cht = BooleanVar(value=False)
chb = ttk.Checkbutton(text='взрослый', variable=cht)
chb.grid(column=0, row=3)
```

Для того, чтобы было удобно оперировать с флажком, который имеет 2 состояния: поднят и опущен, используется специальная переменная, аналогичная `StringVar`, но хранящая логическое значение. Теперь немного модифицируем функцию `swrite`, чтобы она дополняла подпись словом «взрослый», если флажок поднят (активен) или словом «ребёнок», если опущен (неактивен):

```
def swrite(event):
    fname = fd.asksaveasfilename()
    f = open(fname, 'a')
    if cht.get():
        sb = 'взрослый'
    else:
        sb = 'ребёнок'
    f.write(strF.get()+' '+sb+'\n')
    f.close()
```

Как видно из примера, метод `get` для переменной типа `BooleanVar` выдаёт логическое значение, так же как с помощью метода **`set`** туда можно поместить либо `True`, либо `False`.

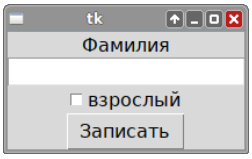


Рис. 11.6. Форма `Tk`, текстовое поле `ttk.Entry`, кнопка `ttk.Button`, метка `ttk.Label`, флаг `ttk.CheckButton`.

Если есть более двух состояний, из которых нужно выбрать, например, не только взрослый/ребёнок, но ещё и пенсионер, `CheckButton` уже не подходит и нужно использовать другой тип виджетов, которые по-русски так и называются «радиокнопка» `Radiobutton`. Эти виджеты всегда используются группами, причём в каждый конкретный момент активным в группе может быть только один. Это достигается тем, что все они соединяются с одною и тою же переменною типа `StringVar`. Поскольку радиокнопок много, создавать их имеет смысл циклом, для чего создадим список из значений (в нашем случае из 3 элементов: ребёнок, взрослый, пенсионер):

```
ludi = ['ребёнок', 'взрослый', 'пенсионер']
radiost = StringVar(value=ludi[0])
for i, lud in enumerate(ludi):
    rb = ttk.Radiobutton(text=lud, variable=radiost, value=lud)
    rb.grid(column=0, row=i+2, sticky=W)
```

Здесь мы воспользовались зарезервированными в начале строками под полем ввода. Кроме того, был использован вариант цикла **for** с **enumerate**, потому что он универсален и выдаёт сразу парами значение из списка (в нашем случае мы назвали его `lud`) и его номер `i`. Считается, что использование **enumerate** предпочтительно циклу с **range**, который можно было бы написать на этом месте, поскольку обращение к элементам списка происходит более явно. Кроме того, при обходе по **enumerate** элементы становятся неизменяемыми, потому что каждый раз работа идёт не с самими ими, а с их копиями, хранящимися во введённой переменной, в нашем случае `lud`.

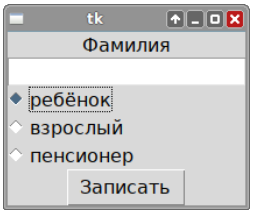


Рис. 11.7. Форма `Tk`, текстовое поле `ttk.Entry`, кнопка `ttk.Button`, метка `ttk.Label`, радиокнопка `ttk.Radiobutton`.

Далее модифицируем снова функцию `swrite`, чтобы она работала с радиокнопками (на самом деле работать она будет только с переменною `radiost`, хранящею текущее выбранное значение):

```
def swrite(event):
    fname = fd.asksaveasfilename()
    f = open(fname, 'a')
    f.write(strF.get()+' '+radiost.get()+'\n')
    f.close()
```

В некотором смысле работа с радиокнопками даже проще, чем с галочкой.

11.3 Списки и меню

В графических приложениях часто необходимо работать с различными списками. Самый простой пример — список строк. Для отображения списка строк можно воспользоваться таким виджетом, как `Listbox`, дословно — коробок со списком. Создадим и разместим такой виджет на приложении:

```
listvar = Variable(value=[])
lbox = Listbox(width=60, height=5, listvariable=listvar)
lbox.grid(column=0, row=0, rowspan=3)
```

Обратите внимание, что виджет `Listbox` находится непосредственно в `tkinter`, а не в `ttk`, и что для его обслуживания также создаётся служебная переменная (у нас называется `listvar`), только не типа `StringVar`, а типа `Variable`. Сам наш виджет будет иметь 5 строк в высоту (что делать, если в его содержимом строк окажется больше, напишем ниже) и 60 символов в ширину. Для обработки списка напишем три кнопки (чтобы они все уместились справа от списка красиво, списку разрешено занимать 3 ряда по вертикали):

```
bts = ttk.Button(text='Считать')
bts.bind('<Button-1>', readf)
bts.grid(column=2, row=0)

btd = ttk.Button(text='Удалить')
btd.bind('<Button-1>', listdel)
btd.grid(column=2, row=1)

btw = ttk.Button(text='Записать')
btw.bind('<Button-1>', writef)
btw.grid(column=2, row=2)
```

Первая кнопка будет считывать данные из файла, открываемого через диалог:

```
def readf(event):
    f = fd.askopenfile()
    for line in f:
        lbox.insert(END, line.strip())
    f.close()
```

Данные читаем построчно и добавляем каждую считанную строку в конец нашего Listbox, чтобы выкинуть возможные начальные и конечные пробелы и символ конца строки, добавляем не саму считанную строку `line`, а её обрезанный с обоих концов вариант `line.strip()`.

Третья кнопка будет записывать изменённый (укороченный) список в новый файл, открываемый на перезапись (если что-то там было, всё стирается):

```
def writef(event):
    f = fd.asksaveasfile()
    for i in range(lbox.size()):
        f.write(lbox.get(i)+'\n')
    f.close()
```

Обратите внимание, что у Listbox есть метод `size`, позволяющий узнать число элементов в нём, и метод `get`, позволяющий получить i-тый элемент в виде переменной стандартного для Python строкового типа по номеру.

Наконец, вторая кнопка позволяет удалить активный элемент из списка:

```
def listdel(event):
    idx = lbox.curselection()
    lbox.delete(idx[0])
```

Здесь использованы ещё два метода Listbox: `curselection` для получения номеров всех выбранных элементов списка — выдаёт кортеж, так как можно специальным свойством `selectmode='multiple'` при создании Listbox выбирать в списке не одни, как у нас, а сразу по нескольку элементов за раз, и `delete` для удаления элемента по номеру.

Рис. 11.8. Форма `Tk`, список `Listbox`, кнопки `ttk.Button`.

В нашем списке есть место всего под пять строк. Но в действительности их может быть и более. Если ничего не делать, `Listbox` позволяет с этим кое-как справляться: прокручивать список можно колесом мыши или клавишами вверх-вниз. Но лучше сделать специальную линейку прокрутки, называемую `Scrollbar`:

```
sbar = ttk.Scrollbar(prog, orient='vertical', command=lbox.yview)
sbar.grid(column=1, row=0, rowspan=3)
lbox.configure(yscrollcommand=sbar.set)
```

Обратите внимание, что полоска прокрутки — такой же виджет, как и сам список. Для его размещения была задействована предусмотрительно оставленная при размещении кнопок первая колонка. Наша полоска прокрутки — вертикальная, что мы упоминаем несколько раз: при ориентации в пространстве (важно для отображения) — `orient='vertical'`, при сообщении полоске, что ей делать — `command=lbox.yview` крутить список `lbox` по координате y, а также при сообщении самому списку, что у него теперь есть вертикальная полоса прокрутки — `lbox.configure(yscrollcommand=sbar.set)`. Заметьте, что при связывании полосы прокрутки и списка мы делаем это в обе стороны: чтобы при прокрутке полосы список двигался и при прокрутке списка клавишами или колесом мыши двигалась полоса. Также важно отметить, что именованные аргументы `command` и `yscrollcommand` суть функции по смыслу. Это значит, что мы передаём одну функцию (в нашем случае вместо функции оба раза используется метод, что часто допустимо), в качестве аргумента другой, не вызывая её — скобок после имени нет. Вызывать переданную функцию будет уже та, которой мы её отдали.

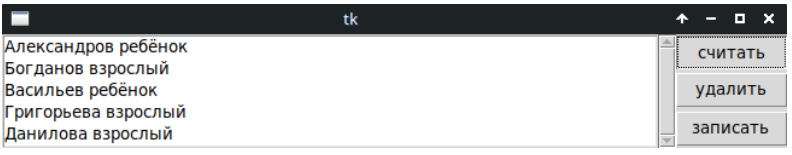


Рис. 11.9. Форма `Tk`, список `Listbox`, кнопки `ttk.Button`, полоса прокрутки `ttk.Scrollbar`.

В любой реальной программе, как правило, есть верхнее, оно же главное меню. Выбирая пункты в нём, можно совершать те или иные действия аналогично нажатию кнопок на форме. Для создания меню в `tkinter` используется виджет типа `Menu`:

```
prog = Tk()
menubar = Menu(prog)
prog['menu'] = menubar
menu_file = Menu(menubar)
menu_edit = Menu(menubar)
```

Обратите внимание, что в `tkinter` меню не является по умолчанию элементом формы, мы должны самостоятельно сообщить при создании меню, кто его родитель. Это может быть как сама форма для `menubar` в приведённом примере, так и другое, родительское меню, как для `menu_file` и `menu_edit`. Во втором случае меню будет выпадающим из пункта меню, то есть иерархически устроенным. Можно сделать больше 2 уровней иерархии: верхний уровень размещается в приложении, второй выпадает вниз, последующие (третий и далее) — вправо. Поскольку меню само по себе не является элементом формы `prog`, не только оно не знает, кто его родитель без прямого сообщения этого, но и родитель `prog` про него тоже ничего не знает, а без этого знания меню на форме не появится. Чтобы сообщить форме, что она содержит меню, приходится использовать не совсем чистый «трюк» `prog['menu'] = menubar`, обращаясь к форме как будто она — словарь. Такой странный подход допустим, поскольку `tkinter` основан на библиотеке `Tk` для языка программирования `Tcl`, в котором всё может быть строкою, значит, и имя параметра — тоже просто строка, а значение по имени можно получить через словарь. Чтобы сообщить родительскому меню о наличии в нём потомков, есть более правильный «питоний» способ:

```
menubar.add_cascade(menu=menu_file, label='Файл')
menubar.add_cascade(menu=menu_edit, label='Правка')
```

Элементы будут размещаться на родительском меню в порядке упоминания при вызове метода `add_cascade`. При размещении меню любого уровня метод `grid` не используется, поскольку никаких столбцов и строк в произвольном порядке занимать они не могут.

Чтобы можно было использовать пункты меню для тех же действий, что и написанные ранее кнопки, при добавлении их можно связать с теми же функциями, например, так:

```
menu_file.add_command(label='Считать из файла', command=readf)
```

Однако тут возникнет проблема: функция `readf` принимает обязательный аргумент–событие `event` (см. код выше), но функция, вызываемая по нажатию на кнопку меню не должна иметь обязательных аргументов, таково требование `tkinter`! В результате приведённый выше код при нажатии на пункт меню будет генерировать ошибку:

```
Exception in Tkinter callback
  Traceback (most recent call last):
    File "/usr/lib/python3.7/tkinter/__init__.py", line 1705, in __call__
return self.func(*args)
TypeError: readf() missing 1 required positional argument: 'event'
```

Есть несколько путей справиться с этою проблемою. Самый примитивный — написать новую функцию `readf2` с тем же содержимым, но без параметров. Это очень плохой путь, потому что он ведёт к дублированию кода. Если в дальнейшем программа будет меняться, код придётся править в обоих местах одновременно и отлаживать дважды. Поэтому первый «вменяемый» вариант — написать функцию–обёртку и подставить её в обработчик меню вместо оригинала:

```
def readf2():
    lego_fio('Бесполезный аргумент')
```

Здесь ничего не делаем, только вызываем исходную функцию `readf` и передаём исходной функции в качестве аргумента бесполезную строку, поскольку аргумент `event` по умолчанию строковый, например, можно вспомнить его значение `'<Button-1>'`, соответствующее нажатию левой кнопки мыши. Можно было бы просто передать `''` или любое иное значение, или даже `None`, ничего бы не изменилось, так как наш аргумент никак не используется в `readf`.

Есть путь попроще: наша функция–обёртка на самом деле нигде, кроме как при определении пункта меню, не нужна. Значит, можно написать неименованную **лямбда–функцию** без аргументов прямо на месте. Используем этот подход для дублирования функционала другой кнопки:

```
menu_edit.add_command(label='Удалить строку', command=lambda: listdel(''))
```

Лямбда функция — это другой по сравнению со стандартными средствами (ключевое слово `def`) способ введения функции. Вызвать такую функцию в произвольном месте программы не получится, так как у неё нет имени, она просто сразу присваивается чему-то (не её результат, а сама функция) или передаётся в качестве параметра другой функции, как в нашем случае. Синтаксис у неё такой: сначала идёт ключевое слово `lambda`, после него безо всяких скобок параметры через запятую, затем двоеточие, затем результат без слова `return`. Очевидно, что записать в виде лямбда функции можно только сравнительно простые выражения, поскольку локальные переменные недопустимы. В приведённом выше примере нам нужна функция без параметров, поэтому сразу после двоеточия идёт результат работы функции, который получается путём вызова уже нормальной, именованной функции `listdel` с аргументом в виде пустой строки.

Рис. 11.10. Форма `Tk`, список `Listbox`, кнопки `ttk.Button`, линейка прокрутки `ttk.Scrollbar`, меню `Menu`.

В действительности, мы только что увидели разницу между **процедурным** и **функциональным программированием**: первая концепция требует явного переопределения или обёртывания функций, вторая — позволяет решать многие проблемы

на месте. То, что функциональное программирование в Python используется не так уж часто, следствие наличия нескольких более мощных дополнительных механизмов, позволяющих решить проблему ещё проще. Вспомним, что приведённое выше сообщение об ошибке жаловалось на наличие одного обязательного позиционного аргумента. В действительности, самый простой способ избавиться от него — сделать этот аргумент необязательным. То есть надо модифицировать заголовок функции `readf` так, чтобы аргумент `event` имел хоть какое-нибудь значение по умолчанию, например, `''` или `'<Button-1>'`. Этого будет достаточно и не нужно писать никаких обёрточных функций или лямбд!

11.4 Холст и рисование

Одним из базовых элементов любого графического интерфейса является **холст** (полотно, canvas) — область, на которой можно рисовать или размещать уже готовые графические объекты. Каждая библиотека виджетов, включая `tkinter`, имеет в своём составе один или несколько вариантов холста. Создадим холст и разместим его на форме:

```
prog = Tk()
canvas = Canvas(prog, width=500, height=500, background='#FFFFFF')
canvas.grid(column=0, row=0)
```

Параметры `width` и `height` означают ширину и высоту холста в точках, `background='#FFFFFF'` значит, что цвет фона будет белый. Вместо `'#FFFFFF'` можно было просто написать `'white'` и это было бы правильно, но приведённый в листинге способ показывает возможность задания произвольного цвета, формируемого из трёх основных цветов, интенсивность которых представлена двузначными шестнадцатеричными цифрами (для отличения шестнадцатеричных чисел от десятичных в начале числа используется символ решётки): `#FF` означает 255, первые два знака соответствуют интенсивности красного, вторые — зелёного, третьи — синего.

Рисование на холсте возможно как со стороны программы, так и со стороны пользователя. В этом холст подобен полю ввода, для которого вы можете изменять значение как из программы, так и позволить это делать пользователю программы. Для рисования средствами программы доступны несколько основных примитивов, из которых можно создавать более продвинутые изображения. Например, нарисовать можно линию и прямоугольник:

```
canvas.create_polygon(150, 200, 250, 100, 350, 200, fill = 'orange')
canvas.create_rectangle(150, 200, 350, 400, fill='blue', outline='grey')
line1 = canvas.create_line(150, 300, 350, 300, width=2)
```

При рисовании линии аргументами метода `create_line` являются координаты начала (первые два числа) и конца (вторые два), именованный параметр `width` позволяет задать ширину линии в пикселях (по умолчанию это 1). Координаты отсчитываются от левого верхнего угла. Можно также сообщить цвет с помощью именованного аргумента, например, чёрный: `fill='black'`, иначе рисование будет производиться текущим цветом. Метод `create_polygon` позволяет рисовать многоугольники, указываются координаты всех углов. В нашем случае координат три, значит в итоге получится треугольник. При рисовании прямоугольника указываются координаты левого верхнего и

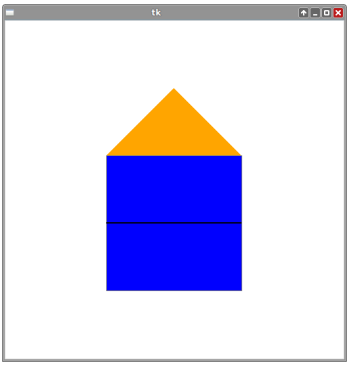


Рис. 11.11. Рисование на холсте `Canvas`.

правого нижнего углов, а кроме цвета заполнения можно также указать цвет внешнего контура `outline='blue'`. Методы, создающие стандартные фигуры, в том числе `create_line` и `create_rectangle` возвращают целое число (номер объекта на холсте), которое можно присвоить какой-нибудь переменной, в приведённом случае — `line1`. Это может быть полезно, если в дальнейшем будет необходимо каким-то образом модифицировать объект. Кроме линий и многоугольников есть ещё целый ряд графических примитивов перечисленных в таблице 11.1.

Таблица 11.1. Основные графические примитивы

Команда	Что рисуется	Число координат	Граница
`create_line`	линия	4	нет
`create_rectangle`	прямоугольник	4	есть
`create_polygon`	многоугольник	6 и более (чётное)	есть
`create_oval`	эллипс/круг	4	есть
`create_arc`	сектор/сегмент/хорда	4	есть
`create_text`	текст	2	нет

Теперь давайте разберёмся с тем, как сделать из нашего полотна простейшую «рисовалку». Для этого добавим две функции:

```
def savePosn(event):
    global lastx, lasty
    lastx, lasty = event.x, event.y
def addLine(event):
    canvas.create_line((lastx, lasty, event.x, event.y), fill=color)
    savePosn(event)
```

Первая функция считывает координаты события, под которым может скрываться любое стандартное событие `tkinter`, например, уже хорошо нам известный щелчок левою кнопкою мыши `"<Button-1>"`. Здесь нам приходится явно указывать, что переменные `lastx` и `lasty` являются глобальными, иначе присвоенные им значения пропадут при выходе из функции. Функция `addLine` также использует координаты последнего события, но уже для того, чтобы нарисовать линию из текущих координат в координаты события.

Переменная `color` также глобальная, её нужно задать в начале программы, желательно сразу после импорта модулей (напишите, например, `color = 'black'`). Объявлять, что она глобальная при помощи идентификатора `global` не нужно, поскольку она используется на чтение. После того, как мы нарисовали линию в новую точку, нужно передвинуть туда текущее положение курсора (что будет, если этого не сделать, вы можете испытать сами). Для этого можно было бы написать такой же код, как и в функции `savePosn`, но проще всего просто вызвать её!

Сделанная нами «рисовалка» заработает, если связать написанные функции с событиями на холсте:

```
canvas.bind("<Button-1>", savePosn)
canvas.bind("<B1-Motion>", addLine)
```

Первую функцию мы связываем с щелчком по холсту, вторую — с передвижением мыши при нажатой левой кнопке — событием `"<B1-Motion>"` (если бы было передвижение при нажатой правой кнопке, было бы `"<B2-Motion>"`).

Теперь пользователь нашей программы может рисовать произвольные линии на холсте. Но только одним и тем же цветом `color`, установленным вначале. Конечно, можно было бы организовать выбор цвета через систему полей ввода, но это уже фактически сделано за нас с помощью специального диалога `askcolor`, для чего нужно импортировать модуль `colorchooser`. Также сделаем кнопку для запуска нужного диалога:

```
from tkinter import colorchooser
def setcolor(event):
    global color
    color0, color1 = colorchooser.askcolor(initialcolor="#000000")
    color = color1
btn = ttk.Button(text='Выбрать цвет')
btn.grid(column=1, row=0)
btn.bind("<Button-1>", setcolor)
```

В качестве значения начального цвета для выбора `initialcolor` лучше всего передавать чёрный цвет. Функция `askcolor` сама запускает диалог и возвращает кортеж, где в двух разных видах: через три действительных числа (это значение попадает в переменную `color0`) и в виде строки (попадает в `color1`) передаётся значение выбранного цвета. Для внутреннего представления в `tkinter` используется строка, поэтому нам полезен первый элемент кортежа, а не нулевой.

Рисование на холсте — это ещё не все возможности. Любой нарисованный объект имеет свой уникальный идентификатор, используя который можно его модифицировать, например, двигать. Вспомним, что выше мы присвоили результат создания линии переменной `line1`. Теперь давайте напишем кнопку, которая будет смещать линию на заранее заданное расстояние:

```
def moveline(event):
    canvas.move(line1, 200, 0)
btm = ttk.Button(text='Передвинуть')
btm.grid(column=0, row=1)
btm.bind("<Button-1>", moveline)
```

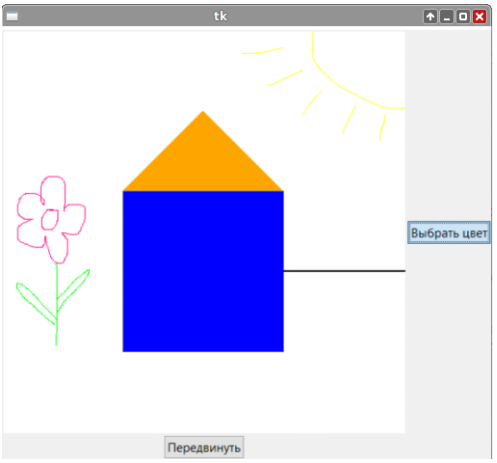


Рис. 11.12. Рисование на холсте `Canvas`.

Естественно, расстояние можно задать из поля ввода.

Кроме перемещения объекта можно делать с ним и другие операции. Например, его можно удалить методом `delete`. Чтобы очистить холст, можно уничтожить в цикле все объекты, либо просто передать в метод `delete` вместо номера объекта параметр `"all"`.

11.5 Принципы объектно-ориентированного программирования

В предшествующих разделах мы познакомились с рядом базовый виджетов. Теперь вернёмся к недописанной программе для создания калькулятора и доделаем её. А заодно рассмотрим, зачем писать собственные классы и как это помогает автоматизировать и сократить код.

Полученная выше программа у нас заняла 150 строк кода. Вообще-то для Python многовато. И это при том, что она умеет только складывать и вычитать. Основная проблема нашей программы — многократное дублирование кода для создания и обработки кнопок. Код создания кнопок легко можно было бы поместить прямо в цикл или функцию и вызвать в цикле, но в таком случае встанет проблема с обработкою, так как каждая функция вроде `renov1` уникальна, поскольку связана со своим символом, а передать унифицированной функции нужный символ в качестве аргумента (вполне разумное желание, ведь за этим аргументы функций и существуют) не получается: ведь вызывать функцию будем не мы, она будет вызываться автоматически при нажатии кнопки.

Возникает мысль, что нужно как-то объединить саму кнопку и её функцию-обработчик в одну сущность так, чтобы функция могла знать, что написано на кнопке (в нашем случае текст на кнопке и является тем самым вводимым символом). Сделать это можно с использованием классов и свойственных им механизмов **наследования** и **инкапсуляции**, то есть основных принципов ***объектно-ориентированного программирования***.

Инкапсуляция представляет собою механизм объединения данных (полей класса) и функций (методов класса) в одном объекте (включения данных и методов объект) так, что все методы знают друг о друге и о полях класса и могут обращаться к полям и друг к другу. Класс при этом является типом данных, а объекты (они называются экземплярами класса) — переменными этого типа. Одновременно в программе могут существовать много различных объектов одного и того же типа (класса), различающиеся значениями полей, то есть все поля присутствуют у них всех, но могут иметь отличаться по значению. Одним из стандартных классов является, например, список, у которого есть много методов.

Наследование — возможность на основе уже существующих классов объектов создавать новые, обладающие дополнительными полями и методами или реализующие некоторые методы иначе (переопределение методов), чем в родительском классе. Наследование позволяет не писать весь код заново каждый раз, а дополнить его в нужной мере. При этом оба класса: и родительский, и дочерний будут пригодны для использования и в программе могут сосуществовать переменные обоих типов. Наследование может быть иерархическим, когда один и тот же класс является родителем по отношению к одним классам и потомком по отношению к другим.

Объявим класс для нашей цифровой кнопки:

```
class DButton(ttk.Button):
```

Этот класс назван нами `DButton`, но назвать его можно как угодно — к именам классов предъявляются те же требования, что и к прочим идентификаторам, то есть к именам переменных и функций. Класс `DButton` является наследником класса `ttk.Button`, как это написано (родительский класс стоит в списке). Python — один из очень немногих языков, позволяющий множественное наследие, т. е. у одного и того же класса может быть несколько родителей, они перечисляются в заголовке через запятую. Но пока мы об этом не будем.

Итак, первое, что нам нужно — это определить инициализатор для объектов — экземпляров класса. Это такая специальная функция, которая вызывается в момент создания переменной этого типа (в Python все переменные — объекты и потому все типы — классы). Инициализатор всегда называется `__init__`:

```
    def __init__(self, symbol):
        super().__init__(text=symbol)
        self.symbol = symbol
        self.bind('<Button-1>', self.renov)
```

Поскольку `__init__` — это функция, то и объявляется она с помощью ключевого слова **`def`**, а также может иметь аргументы. Инициализатор написан с дополнительным сдвигом вправо, это не случайно: все методы нашего класса должны быть написаны с отступом, поскольку так становится ясно, что они вложены в конструкцию `class`. Наш инициализатор имеет 2 аргумента: первый из них имеет имя `self` и это специальный, особенный аргумент, о нём чуть ниже. Второй — имя `symbol`, которое мы придумали сами: это символ (по нашей задумке одна из цифр или точка), обозначающий, что написано на кнопке и для ввода чего она предназначена.

Наш класс — потомок стандартного класса кнопки `ttk.Button`. Но само по себе объявление этого в заголовке недостаточно. При создании кнопки также вызывается её инициализатор. Чтобы наш объект типа `DButton` получил все свойства и мето-

ды кнопки, нужно этот инициализатор вызвать в инициализаторе нашей специальной кнопки. Именно это делает строка `super().__init__(text=symbol)`. Здесь специальная стандартная функция `super()` выдаёт объект — заготовку экземпляра родительского класса, конструктор которого запускается. Поскольку в нашем случае родитель один, ничего указывать в скобках не нужно, достаточно просто пустых скобок, аргументы `super()` нужны на случай, если родителей несколько — это называется *множественное наследование* и такой механизм помимо Python и C++, где он был от основания, теперь реализован в ряде других языков. При запуске инициализатора родительского типа можно передать туда все параметры, что и непосредственно при создании объекта типа `ttk.Button`: в нашем случае это надпись на кнопке: `text=symbol`.

Теперь перейдём к строке `self.symbol = symbol`. Магическое слово `self` — первый аргумент **всех** методов класса — передаёт в методы информацию обо всём классе. То есть через `self` каждый метод может обращаться к любому другому методу или полю (переменной), описанному в классе, если перед его именем поставить `self`. Таким образом, создав объект `self.symbol`, равный значению аргумента конструктора по имени `symbol`, мы как бы сохраняем значение `symbol` в нашем объекте для дальнейшего использования в качестве поля.

Наконец, строка `self.bind('<Button-1>', self.renov)` вызывает метод `bind` нашего класса. Этот метод мы не описывали, но он у нас есть, потому что унаследован от родителя `ttk.Button` и стал доступен после вызова конструктора родителя с помощью `super()`. В качестве функции, которая будет исполняться при нажатии кнопки, передаётся ещё не написанный метод `renov`, чтобы получить доступ к которому впереди его имени нужно поставить `self`.

```
    def renov(self, event):
        stext.set(stext.get() + self.symbol)
```

Метод `renov` имеет два аргумента, но оба они по сути служебные: `self` обеспечивает связь с классом, в составе которого он находится, а `event` нужен для передачи типа действия, на которое будет реагировать кнопка. Сами мы этот метод вызывать никогда не станем, но важно, что в него с помощью `self.symbol` удалось передать символ, написанный на кнопке.

Теперь наш класс готов и можно испытать его в действии. Но для начала заметим, что для массового использования нужно задать координаты всех кнопок для цифр и точки списком:

```
coords = [(0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2),
(0, 3), (1, 3), (2, 3), (1, 4), (2, 4)]
```

Теперь можно создать все кнопки с обработчиками одним циклом:

```
for symbol, coord in zip('1234567890.', coords):
    btn = DButton(symbol=symbol)
    btn.grid(column=coord[0], row=coord[1])
```

Здесь мы проитерировали строку `'1234567890.'`, поскольку её элементы — символы (строки единичной длины), вместе со списком координат. Специальная встроенная функция `zip` даёт возможность итерировать не по одному, а сразу по нескольким составным объектам. Для этого эти объекты помещаются в `zip` как аргументы (`zip` как и

`print` — это функция с переменным числом аргументов), и `zip` выдаёт на каждом шаге цикла кортежи, содержащие по одному элементу из каждого своего аргумента. Так, на первом шаге цикла `symbol` примет значение `'1'`, а `coord` — значение `(0, 1)`, потому что это — нулевые элементы `'1234567890.'` и `coords` соответственно, на втором шаге — значения `'2'` и `(1, 1)` и так далее, пока не закончится самый короткий из включённых в `zip` составных объектов, т. е. число итераций цикла равно длине самого короткого из перечисленных объектов, хвосты остальных игнорируются.

Отметим, что при создании нашей специализированной кнопки мы передали аргумент `symbol` по имени в конструктор, а затем использовали метод `grid`, как будто это простая обычная кнопка `ttk.Button`, потому что наш класс является наследником класса `ttk.Button` и имеет возможность использовать все его методы.

Если теперь удалить все методы и описания кнопок для цифр и точки, заменив их на созданный нами класс, и циклом породить кнопки нового типа, код программы сократится примерно в полтора раза, с примерно 150 до 100 строк.

Кроме кнопок для цифр у нас есть ещё кнопки операций. Их тоже можно унифицировать, поскольку значительная часть кода в них общая. Заодно добавим недостающие операции для умножения и деления:

```
class OButton(ttk.Button):
    """ Класс кнопки для арифметических операций """
    def __init__(self, symbol):
        super().__init__(text=symbol)
        self.symbol = symbol
        self.bind('<Button-1>', self.oper)
    def oper(self, event):
        global hidden, preoper
        if hidden != 0:
            if preoper == '+':
                hidden += float(stext.get())
            elif preoper == '-':
                hidden -= float(stext.get())
            elif preoper == '*':
                hidden *= float(stext.get())
            elif preoper == '/':
                hidden /= float(stext.get())
        else:
            hidden = float(stext.get())
        if self.symbol in ['=', 'C']:
            preoper = ''
            hidden = 0
            if self.symbol == '=':
                stext.set(str(hidden))
            else:
                stext.set('')
        else:
            stext.set('')
            preoper = self.symbol
```

Для выбора операции (помним, что выполняется при нажатии не выбранная, а предыдущая операция, символ которой хранится в глобальной переменной `preoper`) можно воспользоваться конструкцией `elif`. Если в переменной `hidden` хранится 0, это соответствует случаю, когда в памяти ничего не хранится и вводимое число — первое.

Чтобы разместить кнопки операций, также создадим для них список координат `coords2 = [(3, 1), (3, 2), (3, 3), (3, 4), (4, 1),(4, 4)]`, а затем воспользуемся циклом:

```
for symbol, coord in zip('+-*/C=', coords2):
    btn = OButton(symbol=symbol)
    btn.grid(column=coord[0], row=coord[1])
```

В результате код программы сократится ещё примерно на 10 строк при том, что её функционал увеличился: добавились кнопки умножения и деления (см. рис. **??**). Таким образом, изо всех кнопок, сделанных нами без написания отдельного класса, т. е. стандартными средствами, остались только кнопка удаления предыдущего символа и кнопка смены знака.

Об избыточной добродетели — зачем не нужны классы

В настоящее время при обучении многим языкам программирования студентов часто заставляют писать классы направо и налево. В том числе при обучению Python стандартная практика оборачивать каждую программу с графическим интерфейсом в класс. При обучении Java без создания класса вообще нельзя сделать ничего, даже 1+1 сложить (так устроен язык и именно поэтому Java категорически не рекомендуется в качестве первого языка программирования)! На наш взгляд, такое бездумное использование несомненно полезного и мощного инструмента приводит к непониманию, отвращению или безосновательному фанатизму. Классы действительно полезны, но только в ряде случаев. Чтобы мощь классов проявилась, нужно, чтобы программа была достаточно велика. Часто её пишут несколько человек.

Польза от классов несомненна в следующих случаях. Во-первых, если у вас уже есть какой-то тип объектов, который вы хотите наградить новыми дополнительными свойствами и/или методами и писать который с нуля совершенно не хочется. В таком случае будет полезен механизм наследования. Во-вторых, если вам нужно много объектов с разными значениями некоторого свойства, но общими методами, как набор цифровых кнопок в приведённом примере программы–калькулятора. В таком случае класс облегчает серийное создание и управление объектами. В-третьих, если вам нужно построить иерархию объектов, начиная с самого простого, и путём наследования увеличивать сложность. При этом как правило предполагается, что использоваться будут сразу несколько объектов разного уровня сложности. Такой подход позволяет сэкономить время на реализации одинаковых функций для разных объектов.

Нет смысла писать класс просто чтобы избавиться от глобальных переменных, как это частенько предлагается в литературе. Глобальные переменные модуля в принципе могут заменить поля класса, если вы планируете использовать ровно 1 объект. Преимущество классов в таком случае состоит в том, что вы можете реализовать несколько классов в одном файле. Когда вы имеете более одного объекта созданного вами типа без классов обойтись уже будет затруднительно: в таком случае при использовании глобальных переменных значения этих переменных для всех объектов совпали бы, что не

всегда приемлемо. Также следует помнить, что Python предлагает большое число различных способов сделать программы более гибкими, в том числе возможность делать вложенные функции, передавать одну функцию другой в качестве аргумента и использовать в функциях именованные параметры со значением по умолчанию. Во многих других языках аналогичная функциональность достигается только при использовании механизмов наследования и переопределения методов. Часто этих более простых механизмов достаточно, чтобы решить поставленную задачу.

11.6 Примеры решения заданий

Пример задачи 40 (Калькулятор колебательного LC-контура) *Напишите калькулятор для расчёта параметров* LC*-контура.*

Решение задачи 40 Колебательный контур — система, в которой могут происходить свободные (не вынужденные, то есть не обусловленные внешним воздействием) электромагнитные колебания. Простейший контур состоит из катушки индуктивности и конденсатора (это консервативная система, то есть в ней выполняется закон сохранения энергии). Частота, с которой происходят колебания в контуре, называется резонансной. Эту частоту можно изменять, изменяя индуктивность катушки или ёмкость конденсатора. А определяется она формулой Томсона:

$$f = \frac{1}{2\pi\sqrt{LC}},$$

где f — резонансная частота контура в Герцах (Гц), L — индуктивность катушки в Генри (Гн), — ёмкость конденсатора в Фарадах (Ф).

Напишем универсальный калькулятор для расчёта одного из трёх параметров колебательного контура: L, f и C по двум оставшимся, которые вводит пользователь. Пользователь сам может выбирать приставки перед единицами измерения (например, мГн или пФ).

```
import numpy as np
import tkinter as tk
from tkinter import ttk

def calculate(event):
    #Перевод в систему Си
    if Henry.get() == 'нГн': multiH = 1e-9
    elif Henry.get() == 'мкГн': multiH = 1e-6
    elif Henry.get() == 'мГн': multiH = 1e-3

    if Farad.get() == 'пФ': multiF = 1e-12
    elif Farad.get() == 'нФ': multiF = 1e-9
    elif Farad.get() == 'мкФ': multiF = 1e-6

    if Hertz.get() == 'Гц': multiHz = 1
    elif Hertz.get() == 'кГц': multiHz = 1e3
    elif Hertz.get() == 'МГц': multiHz = 1e6
```

```
    if H.get() == '' and F.get()!='' and Hz.get()!='':
        myF = float(F.get())*multiF
        myHz = float(Hz.get())*multiHz
        res = 1/(4*np.pi**2*myHz**2*myF)
        H.set(str(res/multiH))
    elif F.get() == ''and H.get()!='' and Hz.get()!='':
        myH = float(H.get())*multiH
        myHz = float(Hz.get())*multiHz
        res = 1/(4*np.pi**2*myHz**2*myH)
        F.set(str(res/multiF))
    elif Hz.get() == '' and F.get()!='' and H.get()!='':
        myF = float(F.get())*multiF
        myH = float(H.get())*multiH
        res = 1/(2*np.pi*np.sqrt(myH*myF))
        Hz.set(str(res/multiHz))

def cancel(event):
    H.set('')
    F.set('')
    Hz.set('')

prog = tk.Tk()
prog.title('Калькулятор колебательного LC-контура')

#Рисуем колебательный контур
canvas = tk.Canvas(prog, width=500, height=200)
canvas.grid(column=0, row=1, columnspan=3)

canvas.create_line(150, 20, 350, 20, width=2)
canvas.create_line(150, 180, 350, 180, width=2)
canvas.create_line(120, 90, 180, 90, width=2)
canvas.create_line(120, 100, 180, 100, width=2)
canvas.create_line(150, 20, 150, 90, width=2)
canvas.create_line(150, 100, 150, 180, width=2)
canvas.create_line(350, 20, 350, 40, width=2)
canvas.create_line(350, 160, 350, 180, width=2)
canvas.create_arc(335, 40, 365, 70, start = 90, extent = -180,
                  width=2, style = tk.ARC)
canvas.create_arc(335, 70, 365, 100, start = 90, extent = -180,
                  width=2, style = tk.ARC)
canvas.create_arc(335, 100, 365, 130, start = 90, extent = -180,
                  width=2, style = tk.ARC)
canvas.create_arc(335, 130, 365, 160, start = 90, extent = -180,
                  width=2, style = tk.ARC)

#Надписи
```

```
L = ttk.Label(text='Индуктивность катушки L', font=('Arial', 14))
L.grid(column=0, row=2, padx = 20, pady = 20)
C = ttk.Label(text='Ёмкость конденсатора C', font=('Arial', 14))
C.grid(column=0, row=3, padx = 20, pady = 20)
F = ttk.Label(text='Частота колебаний f', font=('Arial', 14))
F.grid(column=0, row=4, padx = 20, pady = 20)

#Текстовые поля
H = tk.StringVar(value='')
H1 = ttk.Entry(textvariable=H, font=('Arial', 14))
H1.grid(column=1, row=2, padx = 20, pady = 20)

F = tk.StringVar(value='')
F1 = ttk.Entry(textvariable=F, font=('Arial', 14))
F1.grid(column=1, row=3, padx = 20, pady = 20)

Hz = tk.StringVar(value='')
Hz1 = ttk.Entry(textvariable=Hz, font=('Arial', 14))
Hz1.grid(column=1, row=4, padx = 20, pady = 20)

#Делаем выпадающие списки
Henry = tk.StringVar()
varHenry = ttk.Combobox(textvariable = Henry, font=('Arial', 14))
varHenry['values'] = ('нГн', 'мкГн', 'мГн')
varHenry.grid(column=2, row=2, padx = 20, pady = 20)
varHenry.current(1)

Farad = tk.StringVar()
varFarad = ttk.Combobox(textvariable = Farad, font=('Arial', 14))
varFarad['values'] = ('пФ', 'нФ', 'мкФ')
varFarad.grid(column=2, row=3, padx = 20, pady = 20)
varFarad.current(1)

Hertz = tk.StringVar()
varHertz = ttk.Combobox(textvariable = Hertz, font=('Arial', 14))
varHertz['values'] = ('Гц', 'кГц', 'МГц')
varHertz.grid(column=2, row=4, padx = 20, pady = 20)
varHertz.current(1)

#Делаем кнопки
btn = ttk.Button(text='Рассчитать')
btn.bind('<Button-1>', calculate)
btn.grid(column=1, row=5)

btnC = ttk.Button(text='Очистить')
btnC.bind('<Button-1>', cancel)
btnC.grid(column=2, row=5)
```

```
prog.mainloop()
```

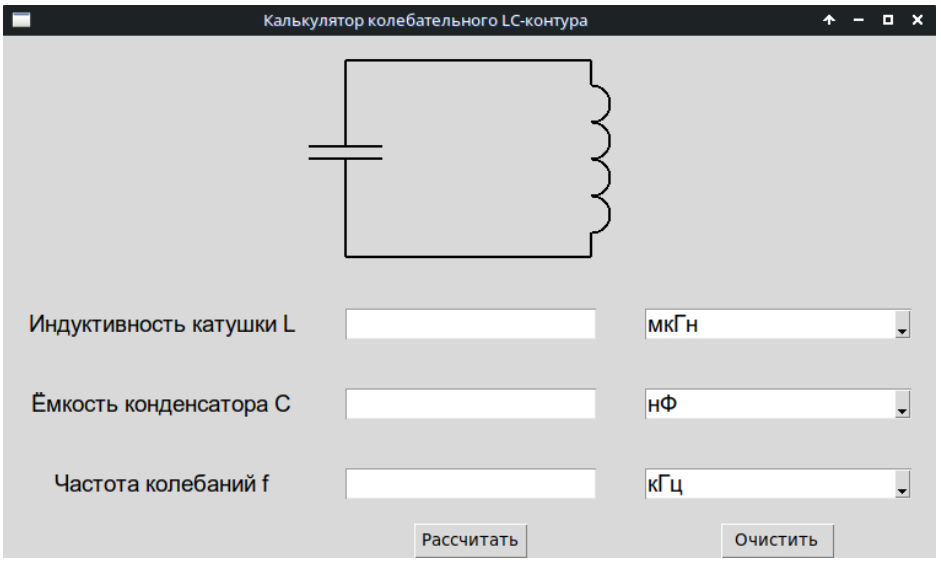


Рис. 11.13. Расчёт колебательного LC-контура. Ёмкость и частота были заданы пользователем, индуктивность рассчитала программа.

Вначале мы наводим «красивости» в нашей программе. С помощью метода `title` напишем на заголовке программы, что она такое: «Калькулятор колебательного LC-контура», а с помощью меток `Label` сделаем поясняющие надписи «Индуктивность катушки L», «Ёмкость конденсатора C», «Частота колебаний f». Дальше с помощью методов виджета `Canvas` нарисуем колебательный контур. Теперь с помощью виджета `Combobox` для каждого параметра сделаем выпадающий список, где можно выбрать подходящий множитель для единицы измерения (пико-, нано-, микро-, мили-, кило-, мега-). Затем расположим на форме виджеты `Entry`, которые помогут вводить необходимые нам значения двух параметров и выводить значение третьего. И, наконец, с помощью виджета `Button` сделаем две кнопки «Рассчитать» и «Очистить».

Теперь можно и функциональную часть нашей программы писать. Все вычисления у нас будут вынесены в две функции `cancel` и `calculate`. Внутри функции `cancel` просто очищаются все три наших текстовых поля. Внутри функции `calculate` вначале все единицы измерения переводятся в систему СИ — пишем какая приставка какому множителю соответствует, а дальше следуют необходимые вычисления в соответствии с формулой Томсона.

Из нового: во всех виджетах прописаны название и размер шрифта (именованный параметр `font`), для метода `grid` добавлены два новых именованных параметра `padx` (отступ справа и слева от виджета по оси x в пикселях) и `pady` (отступ сверху и снизу от виджета по оси y в пикселях). Также использован новый виджет `Combobox` — выпадающий список, его метод `current` позволяет установить одну из строк списка

по умолчанию (мы установили первую строку), а его поле `'value'` позволяет задать возможные варианты приставки для единиц измерения.

11.7 Задания на графический интерфейс

Задание 43 (Ваш личный калькулятор) Используя код из раздела 11.1, соберите программу, реализующую функцию калькулятора, сами и посмотрите, что у вас получится. Что получилось у нас см. на рис. 11.3. Понажимайте на кнопки, убедитесь, что программа не выбрасывает ошибки в терминал.

Задание 44 Задания выполняйте все по порядку.

Сделайте графическую программу из кнопок и полей ввода, которая будет выполнять следующие действия:

1. В поле ввода по умолчанию поставьте 0 (именованный параметр `value`). По нажатию на одну кнопку оно увеличивается на 1, по нажатию на другую — уменьшается на 1.
2. В одном поле ввода пусть будет число, изначально 0. В другом поле ввода другое число, изначально 1. По нажатию на одну кнопку число в первом поле увеличивается на величину числа во втором, по нажатию на другую — уменьшается на ту же величину.
3. В трёх полях ввода три числа. Пусть их стартовые значения будут случайными числами от 1 до 10. По нажатию на кнопку выводить сумму в четвёртое поле ввода.
4. Повторите предыдущую задачу, но теперь не игнорируйте исходное значение в четвёртом поле, а прибавляйте сумму из трёх полей к нему.

Задание 45 Задание на диалоги и радиокнопки. Задания выполняйте все по порядку.

1. Пусть в трёх полях ввода будут имя, фамилия и отчество (изначально поля пустые). Сделайте над ними метки (`Label`), указывающие, что они означают, — например, над полем ввода для фамилии напишите «фамилия». Сделайте программу, которая будет по нажатию кнопки заносить их в файл `ФИО.txt` в виде одной строки по правилу: имя, пробел, отчество, пробел, фамилия (например, `Ольга Петровна Ильина`), используя метод на дозапись (параметр `"a"` вместо `"w"`) так, чтобы введённые записи накапливались в файле. Каждая запись должна начинаться с новой строки (не забудьте добавить символ `\n` в конец).
2. Модифицируйте предыдущую программу, используя файловые диалоги вместо поля ввода. Кроме того, с помощью группы из 5 радиокнопок добавляйте к каждому ФИО ещё одно из слов ребёнок/подросток/взрослый/пенсионер/умер в зависимости от выбора.

Задание 46 Задание на списки и меню. Задания выполняйте все по порядку.

1. Напишите программу, которая по нажатию на кнопку будет визуализировать текстовый файл путём построчной записи его содержимого в `Listbox`.

2. Модифицируйте предыдущую программу, добавив к ней ещё 2 кнопки. По нажатию на одну из них из списка должна удаляться выбранная строка, а по нажатию на другую — всё содержимое `Listbox` будет записано в новый файл. Проверьте работу программы, используя файлы из предыдущего задания, проведя сортировку всех людей на 5 частей по возрасту и записав в 5 разных файлов: детей в один, подростков в другой, взрослый в третий, пенсионеров в четвёртый, умерших в пятый (код для этого писать не нужно, вы должны смочь проделать всё с использованием вашей программы).
3. Добавьте к списку вертикальную полосу прокрутки.
4. Усовершенствуйте программу так, чтобы она позволяла выбирать сразу по несколько элементов в списке и удалять их разом. Обратите внимание, что если не выбран ни один элемент, при попытке удаления в вашей программе может быть ошибка, модифицируйте её так, чтобы этого не было.
5. Усовершенствуйте программу на списки, написанную в предыдущем задании, так, чтобы она позволяла совершать аналогичные изменения как по кнопкам на форме, так и через соответствующие пункты главного меню. Используйте для разных пунктов меню разные механизмы дублирования функционала существующих кнопок: через функции–обёртки, через лямбда–функции и используя значения для параметров по умолчанию.

Задание 47 *Задание на рисование. Задания выполняйте все по порядку.*

1. Напишите простенький графический редактор, на котором будет холст, кнопки для нескольких стандартных цветов (менять цвет основных графических примитивов), поле ввода для выбора ширины линии, кнопка для вызова диалога выбора произвольного цвета и кнопка очистки холста.
2. Модифицируйте предыдущую программу, добавив к ней ещё кнопку, позволяющую отменять последнее действие.
3. В написанной программе есть несколько кнопок, реализующих однотипную функциональность — выбор цвета. Напишите свой класс для этих кнопок путём наследования от стандартного `ttk.Button` подобно тому, как это сделано в примере с калькулятором, и реализуйте вашу программу с использованием данного класса вместо того, чтобы писать все такие кнопки вручную.

Задание 48 (Калькулятор колебательного LC-контура) *Выполнять одно задание с номером* $(n-1)\%m+1$, *где* n *— номер в списке группы, а* m *— число задач в задании.*

Напишите калькулятор для одного из параметров колебательного контура: два параметра вам известны, нужно рассчитать третий. Программа должна выводить на экран название самой программы, метки, текстовые поля, выпадающие списки.

1. Название «Расчёт резонансной частоты». Пользователь заполняет поля «Индуктивность» и «Ёмкость», они идут первыми. В третьей строчке стоит поле «Частота», эта строчка должна заполниться автоматически по нажатию на кнопку «Рассчитать».

2. Название «Расчёт ёмкости». Пользователь заполняет поля «Частота» и «Индуктивность», они идут первыми. В третьей строчке стоит поле «Ёмкость», эта строчка должна заполниться автоматически по нажатию на кнопку «Рассчитать».
3. Название «Расчёт индуктивности». Пользователь заполняет поля «Частота» и «Ёмкость», они идут первыми. В третьей строчке стоит поле «Индуктивность», эта строчка должна заполниться автоматически по нажатию на кнопку «Рассчитать».

Глава 12

Исследование динамических систем средствами Python

Современная теория динамических систем является собирательным названием для нескольких областей исследований, в которых широко используются и эффективным образом сочетаются методы из различных разделов математики: топологии и алгебры, алгебраической геометрии и теории меры, теории дифференциальных форм, теории особенностей и катастроф, теории бифуркаций, а также различных областей вычислительной математики: численных методов решения дифференциальных уравнений, поиска собственных значений и векторов матриц большого размера и др.

Методы теории динамических систем востребованы в других разделах естествознания, таких как неравновесная термодинамика, теория динамического хаоса, теория устойчивости механических систем (устойчивость конструкций, летательных аппаратов и судов, в том числе под воздействием стационарных и нестационарных потоков), синергетика, химическая кинетика, популяционная динамика (например, исследование фито- и зоопланктона).

Роль этой чисто математической дисциплины в современном мире трудно переоценить. Дело в том, что многие результаты, касающиеся поведения нелинейных систем, сегодня получают на основе численного решения модельных уравнений. Компьютерные расчёты, хотя и достаточно точные, тем не менее содержат неизбежные численные погрешности, так же как и результаты натурных экспериментов всегда ограничены ошибками измерений. Значимость численных расчётов существенно возрастает, когда они опираются на результаты строгой аналитической теории, основанной на доказательстве теорем и общих положениях теории динамических систем.

Под **динамической системой** в широком смысле понимают любой объект, эволюционирующий во времени или во времени и пространстве по некоторому детерминированному (от лат. determinans — определяющий) закону — в любой момент времени отсутствуют случайности и шумы, могущие повлиять на будущие значения. В теории динамических систем под динамической системой понимается математическая модель исследуемой физической динамической системы. Математическая модель считается заданной, если известно начальное состояние системы и закон, по которому система переходит из начального состояния в другое (оператор эволюции). Для некоторых классов динамических систем такая постановка задачи: по начальным значениям всех перемен-

ных (исследуемых физических величин) и оператору эволюции рассчитывать поведение в будущем называется «задача Коши». Существуют и другие виды задач, например, с граничными условиями, когда заданы значения не всех динамических переменных, но сразу в нескольких точках, например, в начале и конце временного промежутка. Оператор эволюции в общем случае может быть задан с помощью дифференциальных, интегральных или интегро-дифференциальных уравнений, дискретных отображений последования, а также в форме матриц, графов и т. д. В рамках данного учебника мы ограничимся рассмотрением динамических систем в виде систем с непрерывным временем, которые ещё называются **потоками** (обыкновенных дифференциальных уравнений (ОДУ), дифференциальных уравнений с запаздывающим аргументом, стохастических дифференциальных уравнений (СДУ)) и в виде систем с дискретным временем, которые ещё называют **каскадами** (отображения последования). Динамические системы, которые могут быть использованы для тестирования различных запрограммированных вами методов, размещены в Приложении.

Для наглядного представления информации об исследуемой динамической системе используются пространство состояний (фазовое пространство), пространство параметров и различные комбинированные пространства.

12.1 Численное решение дифференциальных уравнений

Основная задача теории дифференциальных уравнений — **задача Коши**. Она заключается в нахождении решения (интеграла) дифференциального уравнения, удовлетворяющего так называемым начальным условиям. Поэтому иногда говорят «численное интегрирование дифференциальных уравнений», но мы не советуем в случае решения задачи Коши употреблять этот термин во избежании образования в голове «кошицы» — путаницы, когда студенты в одну кучу складывают и методы численного нахождения первообразной и определённого интеграла (методы прямоугольников, трапеций, Симпсона и др.), и численного решения ДУ, например, методы Рунге–Кутты.

Общая задача решения обыкновенных дифференциальных уравнений (**ОДУ**, **ODE** — ordinary differential equation) в численном виде может быть сформулирована следующим образом. Существует векторное дифференциальное уравнение первого порядка:

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}, t), \tag{12.1}$$

где $\vec{x}$ есть вектор, который в дальнейшем, используя терминологию из теории колебаний и нелинейной динамики, мы станем называть **вектором состояния**, состоящий из компонентов $\vec{x} = (x_1, x_2, ..., x_D)$, D — его размерность, t — скалярная независимая переменная (часто — время), $\vec{f}$ — векторная функция той же размерности D, что и вектор состояния $\vec{x}$, состоящая из компонентов $f_i(x_1, x_2, ..., x_D)$, $i = 1, 2, ..., D$.

Векторное уравнение (12.1) может быть рассмотрено как система из D скалярных связанных дифференциальных уравнений (12.2):

$$\begin{aligned} \frac{dx_1}{dt} &= f_1(x_1, x_2, ..., x_D, t), \\ &... \\ \frac{dx_D}{dt} &= f_D(x_1, x_2, ..., x_D, t). \end{aligned} \tag{12.2}$$

Кроме самого уравнения (12.1) заданы также **начальные условия** (12.3), в таком случае принято говорить о решении задачи Коши.

$$\vec{x}(t_0) \quad = \quad \vec{x}_0. \tag{12.3}$$

Требуется найти значения вектора $\vec{x}$ при $t \in [t_0, t_{fin}]$, или, как принято говорить в теории колебаний, необходимо найти траекторию системы на временном промежутке $t \in [t_0, t_{fin}]$, если t рассматривать как время.

12.1.1 Метод Эйлера

Метод Эйлера, также называемый методом Рунге–Кутты первого порядка, — простейший алгоритм численного решения дифференцирования. Идея метода Эйлера — переход от бесконечно малых разностей в (12.1) к конечным малым разностям следующим способом:

$$\frac{\vec{x}(t_{n+1}) - \vec{x}(t_n)}{t_{n+1} - t_n} \approx \vec{f}(\vec{x}_n, t_n). \tag{12.4}$$

Такое приближение справедливо, если величина **шага интегрирования** $\Delta t = t_{n+1} - t_n$ достаточно мала. Что означает «достаточно» — необходимо выяснять в каждом конкретном случае (для каждой системы уравнений) отдельно, общего правила на этот счёт не существует. Однако можно рекомендовать брать Δt много меньше характерного временного масштаба изменений $\vec{x}$ (иногда такой масштаб легко оценить аналитически или из дополнительных соображений), либо наименьшего из таких масштабов, если их несколько.

Из формулы (12.4) легко получить рекуррентное соотношение для расчёта $\vec{x}(t_{n+1})$ через $\vec{x}(t_n)$:

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \vec{f}(\vec{x}_n, t_n)\Delta t. \tag{12.5}$$

Фактически, численное решение дифференциальных уравнений методом Эйлера сводится к замене системы (12.1) соотношением (12.5). Вы подставляете ваши начальные условия (12.3) в (12.5) и получаете значение на следующем шаге, подставляете в (12.5) уже его и т. д. В итоге, получается траектория системы, т. е. значения $\vec{x}$ во все последующие моменты времени с шагом Δt.

Давайте методом Эйлера решим систему дифференциальных уравнений (А.18) третьего порядка ($D = 3$), известную как система Лоренца:

$$\begin{cases} \dfrac{dx}{dt} = \sigma(y - x), \\ \dfrac{dy}{dt} = x(r - z) - y, \\ \dfrac{dz}{dt} = xy - bz, \end{cases}$$

где $\sigma = 10$, $r = 28$, $b = 8/3$. Систему будем решать на временном промежутке $t \in [0, 100]$ с шагом интегрирования $\Delta t = 0.01$, начальные условия зададим $x_0 = 0.01$, $y_0 = 0.02$, $z_0 = 0.03$.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def Lorenz(x, y, z): #Система Лоренца
    sigma = 10
    r = 28
    b = 8/3
    dx = sigma*(y-x)
    dy = x*(r-z)-y
    dz = x*y-b*z
    return dx, dy, dz

dt = 0.01 #шаг интегрирования
N = 10000
t0 = np.arange(0, N*dt+dt, dt) #ряд времени

x0 = np.zeros(N+1)
y0 = np.zeros(N+1)
z0 = np.zeros(N+1)
x0[0] = 0.01; y0[0] = 0.02; z0[0] = 0.03 #начальные условия

for i in range(N):
    dx, dy, dz = Lorenz(x0[i], y0[i], z0[i])
    x0[i+1] = x0[i] + (dx * dt)
    y0[i+1] = y0[i] + (dy * dt)
    z0[i+1] = z0[i] + (dz * dt)

plt.subplot(3, 1, 1)
plt.plot(t0, x0, color = 'black')
plt.title('Временная реализация колебаний системы Лоренца')
plt.ylabel('x')

plt.subplot(3, 1, 2)
plt.plot(t0, y0, color = 'black')
plt.ylabel('y')

plt.subplot(3, 1, 3)
plt.plot(t0, z0, color = 'black')
plt.xlabel('t')
plt.ylabel('z')
plt.tight_layout()
plt.show()
```

В итоге можно нарисовать временные реализации всех трёх переменных системы Лоренца (рис. 12.1).

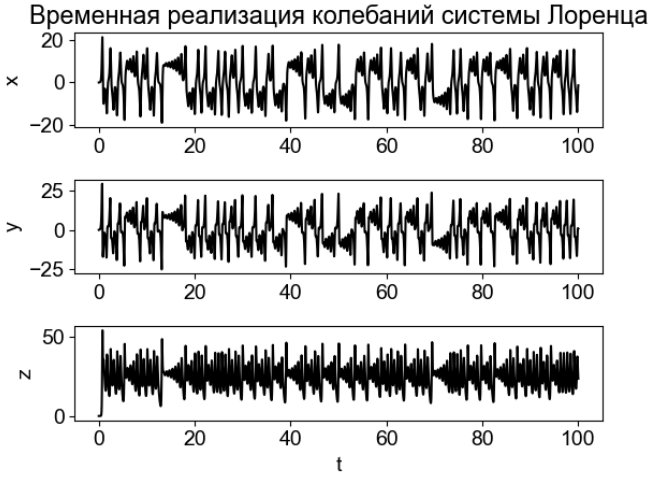


Рис. 12.1. Временная реализация колебаний системы Лоренца.

12.1.2 Метод Рунге-Кутты 4-го порядка для одного уравнения первого порядка

Метод Рунге–Кутты четвёртого порядка — наиболее распространённый метод из семейства методов Рунге–Кутты. Объяснять методы Рунге–Кутты порядков выше первого лучше сначала на одном скалярном уравнении, а не на системе уравнений, как это было сделано выше. Пусть имеется уравнение 1-го порядка общего вида (12.6) и начальные условия $x(t_0) = x_0$ (обычно принимают $t_0 = 0$, но это не строго обязательно).

$$\frac{dx}{dt} = f(x, t), \tag{12.6}$$

где $f(x, t)$ — произвольная непрерывная функция. Метод Эйлера использует для получения значения в $(n+1)$-ый момент времени значение функции f в момент времени t_n и значение координаты в этот момент x_n. Очевидно, однако, что такое рассмотрение является сильно несимметричным: на всём временном интервале от t_n до t_{n+1} динамика по сути игнорируется и функция $f(x, t)$ считается состоящей из полочек. Такой подход напоминает метод прямоугольников при численном интегрировании или аппроксимацию полиномом нулевого порядка — константой. Очевидно, что такая аппроксимация — очень грубая и можно придумать более точную аппроксимацию, что и было сделано Рунге и Куттой.

Метод 4-го порядка основан на построении 4-х приближений для функции f:

$$\begin{aligned} k_1(n) &= f(x_n, t_n), \\ k_2(n) &= f\left(x_n + \frac{\Delta t}{2} k_1, t_n + \frac{\Delta t}{2}\right), \\ k_3(n) &= f\left(x_n + \frac{\Delta t}{2} k_2, t_n + \frac{\Delta t}{2}\right), \\ k_4(n) &= f\left(x_n + \Delta t k_3, t_n + \Delta t\right), \end{aligned} \tag{12.7}$$

где введено обозначение $\Delta t = t_{n+1} - t_n$ — шаг интегрирования, который может совпадать с интервалом дискретизации или быть кратно меньше, также может быть переменным $\Delta t_n = t_{n+1} - t_n$, тогда метод называется методом с переменным или адаптивным шагом, или постоянным, как в приведённой формуле (12.7), тогда метод называется методом с фиксированным шагом.

Если рассмотреть систему (12.7) внимательнее, то видно, что k_1 по сути есть то же приближение, что используется в методе Эйлера, k_2 и k_3 суть два последовательных приближения в середине временного интервала, основанные на аппроксимации динамики, полученной в предыдущем приближении, а k_4 — приближение на конце интервала, основанное на всех предыдущих приближениях рекурсивно. В качестве итеративной формулы для вычисления x_{n+1} при известном x_n используют обычно следующую взвешенную сумму из всех четырёх приближений $k_i(n)$:

$$x_{n+1} = x_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{12.8}$$

Формула (12.8) фактически аналогична формуле (12.5), если отметить, что в новой формуле при аппроксимациях функции стоят веса $\frac{1}{6}$, $\frac{1}{3}$, $\frac{1}{3}$ и $\frac{1}{6}$, а в исходной на их месте стояли 1, 0, 0 и 0 (сумма весов всегда должна равняться 1). В действительности, можно построить сколько угодно методов Рунге–Кутты 4-го порядка, меняя веса так, чтобы они были неотрицательными числами, сумма которых равна 1 (говорят о трёхпараметрическом семействе методов), но чаще всего используется именно классический метод с весами как в формуле (12.8).

Чтобы продемонстрировать, что метод Рунге–Кутты 4-го порядка даёт существенно более точное решение, чем метод Эйлера, решим обоими методами уравнение Ферхюльста (A.1) и наложим наши решения на аналитическое, которое можно получить по формуле (A.2).

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

r = 1.
K = 100.
dt = 0.25

def verhulst(x): # один шаг уравнения Ферхюльста
```

```
    return r*x*(1-x/K)

def vh_euler(x0, N): # Решает Ферхюльста методом Эйлера
    x = x0
    series = [x0]
    for i in range(N-1):
        x += dt * verhulst(x)
        series.append(x)
    return series

def vh_rk4(x0, N): # Решает Ферхюльста методом Рунге-Кутты 4-го порядка
    x = x0
    series = [x0]
    for i in range(N-1):
        k1 = verhulst(x)
        k2 = verhulst(x+dt/2*k1)
        k3 = verhulst(x+dt/2*k2)
        k4 = verhulst(x+dt*k3)
        x += dt/6 * (k1 + 2*k2 + 2*k3 + k4)
        series.append(x)
    return series

x0 = 1.
N = 40
eser = vh_euler(x0, N)
rk4ser = vh_rk4(x0, N)
tser = np.linspace(0, dt*N, N, False)
aser = K*x0*np.exp(r*tser) /(K + x0*(np.exp(r*tser)-1))
plt.plot(tser, eser, '-', color='black', label='Метод Эйлера')
plt.plot(tser, rk4ser, color='gray', label='Метод Рунге-Кутты 4')
plt.plot(tser, aser, 'x', color='black', label='Аналитическое решение')
plt.title('Временная реализация модели Ферхюльста')
plt.xlabel(r'$t$')
plt.ylabel(r'$x$')
plt.legend(loc='best')
plt.show()
```

Из рис. 12.2 видно, что решение методом Рунге–Кутты 4-го порядка практически идеально ложится на аналитический результат в то время как решение методом Эйлера при том же шаге интегрирования заметно от него отличается. С уменьшением шага интегрирования различие будет постепенно исчезать, но это приведёт к увеличению вычислительных затрат на получение решения на том же временном промежутке, так как придётся делать больше шагов. Для сложных систем из большого числа уравнений, а также в задачах, требующих решения систем в реальном времени (например, системы автопилота, коррекции курса корабля и иные задачи оптимального управления), это часто бывает существенно. К тому же метод Эйлера имеет точность порядка $O(\Delta t)$, то есть при уменьшении шага вдвое погрешность относительно аналитического решения уменьшается в среднем тоже вдвое, а метод Рунге–Кутты 4-го порядка имеет

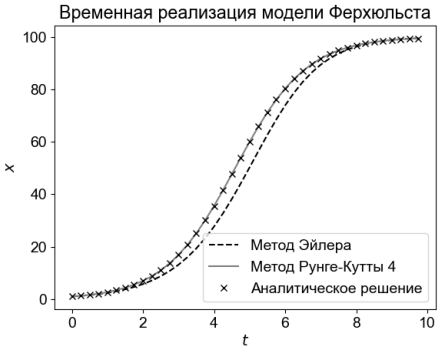


Рис. 12.2. Логистическая кривая — решение уравнения Ферхюльста (A.1) при $r = 1$, $K = 100$, $\Delta t = 0.25$, $x_0 = 1$ методом Эйлера (штриховая чёрная линия), методом Рунге–Кутты 4-го порядка (сплошная серая линия) и аналитическое (крестиками).

точность $O((\Delta t)^4)$, то есть при уменьшении шага вдвое погрешность падает в 16 раз, что позволяет добиться машинной точности во многих задачах при разумной величине шага.

12.1.3 Метод Рунге–Кутты 4-го порядка для системы уравнений

Для системы уравнений метод строится в целом аналогично тому, как он был построен для отдельного уравнения, однако не только переменная x и функция f, но и коэффициенты k_1, k_2, k_3 и k_4 становятся векторами (массивами), длина которых равна числу уравнений. Продемонстрируем метод на примере модели Лотки–Вольтерра A.16 (хищник–жертва), которая часто рассматривается в качестве обобщения уравнений Ферхюльста. Решение приведено ниже в виде листинга (операторы импорта те же, что и ранее и потому опущены для краткости):

```
def lotka(x, p):
""" один шаг уравнения Лотки-Вольтерра """
    dx = np.zeros(2)
    dx[0] = (p['alpha'] - p['beta']*x[1]) * x[0]
    dx[1] = (-p['gamma'] + p['delta']*x[0]) * x[1]
    return dx

def lotka_rk4(x0, N, p):
""" Решает Лотки-Вольтерра методом Рунге-Кутты 4-го порядка """
    x = x0
    series = [x0]
    for i in range(N-1):
        print(x)
```

```
        k1 = lotka(x, p)
        k2 = lotka(x+dt/2*k1, p)
        k3 = lotka(x+dt/2*k2, p)
        k4 = lotka(x+dt*k3, p)
        x = x + dt/6 * (k1 + 2*k2 + 2*k3 + k4)
        series.append(x)
    return np.array(series)

p = {'alpha': 0.9, 'beta': 0.1, 'gamma': 0.8, 'delta': 0.2}
dt = 0.25
x0 = [10., 21]
N = 200
rk4ser = lotka_rk4(x0, N, p)
tser = np.linspace(0, dt*N, N, False)
plt.plot(tser, rk4ser[:, 0], color='gray', label='x - жертва')
plt.plot(tser, rk4ser[:, 1], '-', color='black', label='y - хищник')
plt.title('Временная реализация модели Лотки-Вольтерра')
plt.xlabel(r'$t$')
plt.ylabel(r'$x, y$')
plt.legend(loc='best')
plt.show()
```

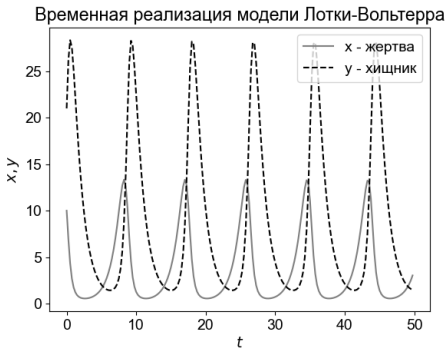


Рис. 12.3. Временные реализации модели Лотки-Вольтерра. Серая линия — численность жертв, чёрная линия — численность хищников.

За счёт того, что Python может работать с массивами почти как с числами, изменения не очень заметны, но нужно обратить внимание на следующие важные особенности. В строчке, где вычисляется новое значение `x` мы заменили оператор добавления `+=` оператором присваивания `=`, так как теперь `x` — массив. Массив в отличие от числа — сложный объект, операции с которым происходят по ссылке. При помещении составного объекта в список его содержимое не копируется, а создаётся новая ссылка. Значит, если мы будем использовать `+=`, новые массивы не будут возникать, а будут генериро-

ваться только ссылки на одни и те же, модифицируемые в процессе работы программы данные, то есть в конце все элементы списка содержали бы одни и те же значения, соответствующие последнему моменту времени. В нашем случае было необходимо принудительно создать новый массив на каждом шаге, что и делает использованная операция, складывая два массива. Также мы передали параметры в уравнение в виде словаря `p`.

12.1.4 Решение ОДУ с помощью функции `scipy.integrate.odeint`

В модуле `scipy` подмодуле `integrate` есть функции `odeint` для решения систем ОДУ. В качестве обязательных параметров эта функция принимает три параметра: функцию, описывающую наше уравнение (функция должна вычислять значения производных, используя выражения справа, и выдавать список или массив, длина которого равна порядку модели D), начальные условия (список или массив значений начальных условий по одному для каждой переменной) и массив из N моментов времени, в которые требуется получить решение. Функция возвращает двумерный массив размера $N \times D$ (N строк, соответствующих разным моментам времени, и D столбцов, соответствующих различным динамическим переменным).

Решим с помощью этой функции систему Рёсслера (А.17) и построим временные реализации всех трёх координат (рис. 12.4):

$$\begin{cases} \dfrac{dx}{dt} = -(y+z), \\ \dfrac{dy}{dt} = x + ay, \\ \dfrac{dz}{dt} = b + z(x-c), \end{cases}$$

где a, b, c — положительные константы. Будем использовать значения параметров, которые использовал сам Рёсслер: $a = b = 0.2$, $c = 5.7$.

```
from scipy.integrate import odeint
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def func(y, t):
    dy = np.zeros(3)
    a = b = 0.2
    c = 5.7
    dy[0] = -y[1]-y[2]
    dy[1] = y[0]+a*y[1]
    dy[2] = b+y[2]*(y[0]-c)
    return dy

y0 = 0.001*np.ones(3) # Начальные условия
t0 = np.linspace(0, 500, 10001)
X = odeint(func, y0, t0)
```

```
plt.subplot(3, 1, 1)
plt.plot(t0, X[:,0], color = 'black')
plt.title('Временная реализация колебаний системы Рёсслера')
plt.ylabel('x')
plt.subplot(3, 1, 2)
plt.plot(t0, X[:,1], color = 'black')
plt.ylabel('y')
plt.subplot(3, 1, 3)
plt.plot(t0, X[:,2], color = 'black')
plt.xlabel('t')
plt.ylabel('z')
plt.tight_layout()
plt.show()
```

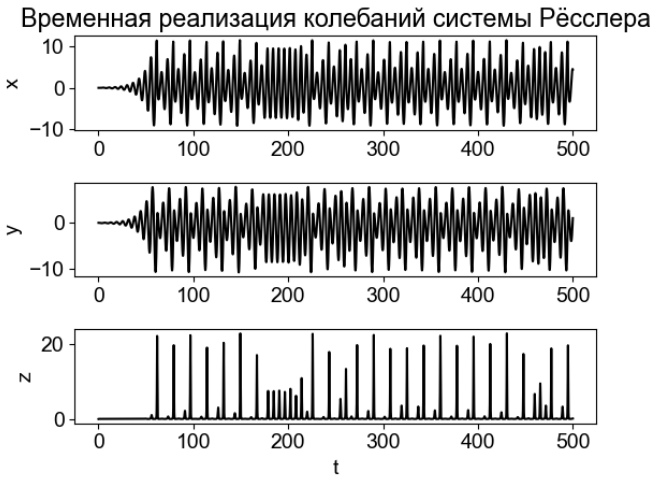


Рис. 12.4. Временная реализация колебаний системы Рёсслера.

12.1.5 Решение ОДУ с помощью функции `scipy.integrate.solve_ivp`

На данный момент функция `odeint` из модуля `scipy.integrate` рекомендуется не для всех типов уравнений, так как разработана новая, более общая функция — `scipy.integrate.solve_ivp` (на самом деле под этим названием скрывается обёртка сразу к нескольким алгоритмам решения, выбор которых возможен с помощью необязательных аргументов). Попробуем с помощью этой функции решить ещё одну трёх-

мерную систему — модель нейрона Хиндмарш–Роуз (A.24):

$$\begin{cases} \dfrac{dx}{dt} = y - ax^3 + bx^2 - z + I_a, \\ \dfrac{dy}{dt} = c - dx^2 - y, \\ \dfrac{dz}{dt} = r\big(s(x - x_R) - z\big), \end{cases}$$

где $a = 1$, $b = 3$, $c = 1$, $d = 5$, $r = 10^{-3}$, $s = 4$, $x_R = -1.6$, $I_a = 1.5$.

```
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def func(t, y):
    a = 1; b = 3; c = 1; d = 5
    r = 1e-3; s = 4; xR = - 1.6; Ia = 1.5
    dy = np.zeros(3)
    dy[0] = y[1]-a*y[0]**3+b*y[0]**2-y[2]+Ia
    dy[1] = c-d*y[0]**2-y[1]
    dy[2] = r*(s*(y[0]-xR)-y[2])
    return dy

y0 = 0.001*np.ones(3)
X = solve_ivp(func, (0, 2000), y0 , method = 'RK45')

plt.plot(X.t, X.y[0, :], color = 'black')
plt.grid()
plt.title('Временная реализация колебаний \nсистемы Хиндмарш-Роуз')
plt.xlabel('t')
plt.ylabel('X')
plt.show()
```

Итак, в качестве обязательных параметров эта функция принимает три параметра: функцию, описывающую наше уравнение (обратите внимание, что в отличие от `odeint` параметры этой функции идут в другом порядке), кортеж из двух чисел: начальный и конечный моменты времени и начальные условия. Из необязательных параметров мы в этом примере использовали параметр `method` — метод решения (численная схема). Он может принимать следующие значения: `'RK45'` — явный метод Рунге–Кутты 4-го порядка (стоит по умолчанию), `'RK23'` — явный метод Рунге–Кутты 3-го порядка, `'DOP853'` — явный метод Рунге–Кутты 8-го порядка (используется аппроксимирующий полином 7-го порядка, можно использовать для уравнений в комплексных числах), `'Radau'` — неявный метод Рунге-Кутты 5-го порядка, шаг задаётся автоматически и точность контролируется с использованием разложения в ряд вплоть до третьего члена включительно, кубический полином используется для интерполяции значений между шагами при необходимости, `'BDF'` — неявный метод переменного порядка (от 1 до

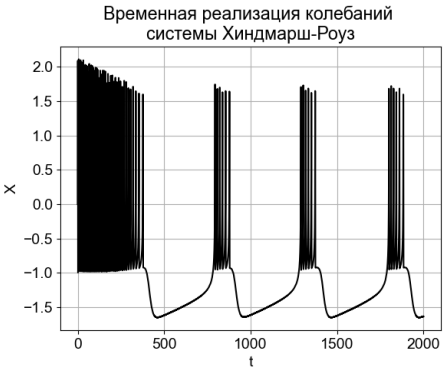


Рис. 12.5. Временная реализация колебаний переменной x системы Хиндмарш-Роуз.

5), для аппроксимации производной применяется формула «обратного дифференцирования», используется переменный (квазипостоянный) шаг, может применяться для уравнений с комплексными величинами, `'LSODA'` — метод Адамса (обобщение методов Рунге–Кутты 5-6 порядков) с автоматическим определением шага. Функция возвращает запись (объект), основными полями которого являются моменты времени (поле `.t`) и само решение (поле `.y`), также имеется большое число дополнительных полей, несущих служебную информацию и информацию о возможных проблемах. Обратите внимание, что решение устроено не так, как в `odeint`: временные ряды переменных будут расположены в нём не по столбцам, а по строкам.

12.1.6 Решение дифференциальных уравнений с запаздыванием

Дифференциальные уравнения с запаздыванием (**ДУ с запаздыванием**, **DDE** — delay differential equation) — вид моделей, более общих, чем обыкновенные дифференциальные уравнения, и в то же время являющиеся подклассом (упрощённым случаем) дифференциальных уравнений в частных производных. Вектор состояния ДУ с запаздыванием бесконечномерный, как для систем в частных производных, потому что будущее зависит не только от состояния системы в один конкретный момент времени, но от её состояния на некотором временном отрезке (а значений на отрезке бесконечно много), но в то же время сам оператор эволюции описывается в терминах обыкновенных производных. Популярность ДУ с запаздыванием связана с рядом факторов: они появляются естественным образом в задача нейродинамики, телекоммуникации и связи из-за наличия запаздывания в канале связи или синапсе.

В общем виде ДУ с запаздыванием можно записать в следующем виде:

$$\frac{dx}{dt} = f(x(t), x(t-\tau)), \tag{12.9}$$

где $x(t-\tau)$ — решение системы в прошлом.

Попробуем решить уравнение генератора с запаздыванием с квадратичной нелинейностью (A.29):

$$\varepsilon \frac{dx}{dt} = -x(t) + \lambda - x^2(t - \tau_0),$$

где $\varepsilon = 20$, $\lambda = 1.05$, $\tau_0 = 100$.

ДУ с запаздыванием можно решать только методами Рунге-Кутты 1-го или 2-го порядков.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def Delay(x, xtau): #Генератор с запаздыванием с квадратичной нелинейностью
    eps = 20
    l = 1.05
    dx = (-x+l-xtau**2)/eps
    return dx

dt = 0.01 #шаг интегрирования
realtau = 100
tau = round(realtau/dt)
realN = 3000
N = round(realN/dt)

t0 = np.linspace(0, realN, N, False) #ряд времени

X = [0.001]*(tau+1) #заполняем первые tau+1 начальных значений

for i in range(tau,N-1): #решаем методом Эйлера
    dx = Delay(X[i],X[i-tau])
    X.append(X[i] + dx*dt)

plt.plot(t0, X, color = 'black')
plt.title('Временная реализация генератора с запаздыванием \n
          с квадратичной нелинейностью')
plt.ylabel('x')
plt.xlabel('t')
plt.show()
```

12.1.7 Решение стохастических дифференциальных уравнений

Если на динамику модели влияет любая случайная величина (**шум**), такую модель принято называть **стохастической** вне зависимости от того, каков вклад шумовой добавки. На практике такое определение вызывает большие затруднения при изложении многих вопросов, поскольку роль шума может быть очень различна и будет зависеть от

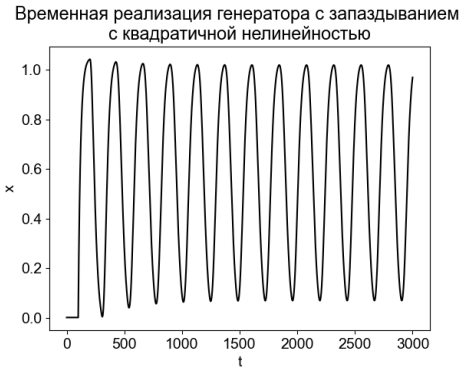


Рис. 12.6. Временная реализация генератора с запаздыванием с квадратичной нелинейностью.

его интенсивности, коррелированности, закона распределения, способа внесения и очень значительно от свойств той системы, на которую шум влияет. Полезно понимать, что в большинстве случаев имеются две части оператора эволюции: динамическая и стохастическая, относительный вклад которых может быть различен. Для многих автоколебательных систем внесение небольших шумов не приводит к качественному изменению поведения, лишь немного искажая решение. Для ряда систем, например, для линейного осциллятора, внесение шума приводит к изменению типа поведения: появляются новые колебательные режимы (к этому же классу событий следует отнести явление стохастического резонанса в нелинейных бистабильных системах), но многие характеристики появившегося под влиянием шума режима, например, частота колебаний, определяются не свойствами шума, а свойствами динамической части оператора эволюции — шум выступает главным образом источником энергии. Таким образом, даже при наличии шума изучение стохастических моделей методами теории динамических систем представляется оправданным. Именно такие модели будут рассмотрены в данном разделе в отличие от систем (**случайных процессов**), для которых шум является основным в их динамике.

Для описания случайных процессов в непрерывном времени используют стохастические дифференциальные уравнения (**СДУ**, **SDE** — stochastic differential equation). Физики обычно предпочитают записывать СДУ в форме уравнения Ланжевена:

$$\frac{dx}{dt} = f(x,t) + g(x,t)\xi(t),$$

где f, g — гладкие функции, ξ — шумовой процесс. Причём часто для простоты это уравнение пишется при $g = const$:

$$\frac{dx}{dt} = f(x(t)) + \xi(t). \tag{12.10}$$

При такой записи хорошо видно, что СДУ состоит из двух частей: динамической ($f(x(t))$) и стохастической ($\xi(t)$).

Численное решение СДУ представляет собою гораздо более сложную задачу, чем численное решение ОДУ, так как при фиксированном начальном условии $x(t)$ СДУ определяют не единственную будущую траекторию системы, а целый ансамбль возможных траекторий, да и интеграл от случайного процесса $\xi(t)$ является сложным понятием. Наиболее простой подход для численного решения СДУ — использование метода Эйлера с малым шагом интегрирования dt. Разностная схема для метода Эйлера (12.5) в случае решения СДУ будет несколько отличаться:

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \vec{f}(\vec{x}_n, t_n)\Delta t + \vec{\xi}(t_n)\sqrt{\Delta t}. \tag{12.11}$$

Внимательные читатели должны были сразу увидеть принципиальное отличие формулы (12.11) от формулы (12.5): умножение стохастического слагаемого не на шаг интегрирования, а на квадратный корень из него $\sqrt{\Delta t}$. Если предположить, что временная реализация, полученная в результате решения уравнения без шума, есть непрерывная функция времени (достаточно общее предположение, выполняющееся даже если сами входящие в уравнения функции имеют разрывы первого рода), то при стремлении $\Delta t \to 0$ в разностной схеме будет выполняться $x(t_{n+1}) - x(t_n) = \Delta x \to 0$. Однако вклад шума не зависит от шага интегрирования и будет фиксированным при $\Delta t \to 0$, то есть не будет стремиться к 0. Таким образом, при достаточно малых шагах интегрирования вся динамика будет определяться исключительно шумом, что абсурдно. Следовательно, необходимо перенормировать шум, привязав его дисперсию к шагу интегрирования, в соответствии с положениями статистической радиофизики, используя его свойства и соответствующие правила интегрирования.

Численно решим уравнение линейного осциллятора с затуханием под воздействием шума (Б.1):

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x = \xi(t),$$

где собственная частота колебаний $\omega_0 = 1$, коэффициент затухания (диссипации) $\gamma = 0.1$, нормальный шум $\xi(t)$ с параметрами $\mu = 0$ (нулевое среднее), $\sigma = 1$ (среднеквадратичное отклонение).

Численные методы позволяют решать только ДУ и системы ДУ первого порядка, разрешённые относительно производных. Это означает, что имеющуюся динамическую систему нужно методом преобразований и замен свести к такому виду, когда слева стоят только первые производные, а справа — произвольные функции от фазовых переменных, но без производных. Будем решать дифференциальное уравнение второго порядка методом замены: для этого обозначим $x = x_0$ и $\dfrac{dx}{dt} = x_1$, тогда уравнение линейного осциллятора можно переписать следующим образом:

$$\begin{cases} \dfrac{dx_0}{dt} = x_1(t), \\ \dfrac{dx_1}{dt} = -2\gamma x_1(t) - \omega_0^2 x_0(t) + \xi(t). \end{cases}$$

из методов, рассмотренных ранее, решать эту систему, можно только методом Эйлера. Возможно и использование методов Рунге–Кутты более высокого порядка, но там формулы будут существенно отличаться от случая ОДУ.

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

```
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def LinNoise(x): #Линейный осциллятор под воздействием шума
    w = 1
    g = 0.1
    dx = np.zeros(2)
    dx[0]= x[1]
    dx[1]=-2*g*x[1]-w**2*x[0]
    return dx

dt = 0.01 #шаг интегрирования
realN = 300
N = round(realN/dt)

t0 = np.linspace(0, realN, N, False) #ряд времени

X = 0.01*np.ones((N,2)) #массив для записи всех значений
for i in range(N-1):
Ksi = np.zeros(2)
Ksi[1] = np.random.normal(0,1) #стохастическая добавка
dx = LinNoise(X[i])
X[i+1] = X[i] + dx*dt + Ksi*np.sqrt(dt)

plt.plot(t0, X[:,0], color = 'black')
plt.title('Временная реализация линейного осциллятора \nпод воздействием шума')
plt.ylabel('x')
plt.xlabel('t')
plt.show()
```

Обратите внимание, что без шума — можно в программе поставить параметры нормального шума (0,0), что будет аналогично отсутствию шума, — колебания такого осциллятора будут затухать, см. рис. 12.7(a). То есть этот тот самый случай, описанный выше, когда внесение шума приводит к качественному изменению поведения системы. Благодаря внесению шума, можно наблюдать колебательный режим, см. рис. 12.7(b).

12.2 Фазовый портрет

Временные ряды — не единственный графический способ представления решения ДУ. Рассмотрим построение фазового портрета динамической системы на примере осциллятора ван дер Поля (A.8):

$$\frac{d^2x}{dt^2} - \left(r - x^2\right)\frac{dx}{dt} + \omega_0^2 x \quad = \quad 0,$$

где ω_0 — параметр, отвечающий за частоту гармонических колебаний (строго говоря, частота будет равна ω_0 только при бесконечно малой амплитуде, а потом будет зависеть

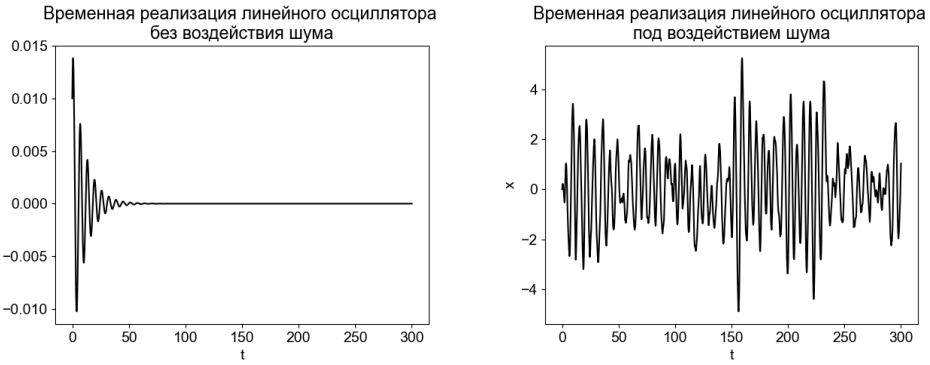


Рис. 12.7. Временные реализации линейного осциллятора: (a) — без шума, (b) — с нормальным шумом.

также и от r), r — параметр нелинейности ($r < 0$ — нет колебаний, $r > 0$ — есть колебания).

Фазовое пространство — пространство, на котором представлено множество всех состояний системы так, что каждому возможному состоянию системы соответствует точка фазового пространства. Размерность фазового пространства зависит от размерности исследуемой системы. Осциллятор ван дер Поля имеет размерность 2, поэтому для построения фазового портрета этой системы нам достаточно двумерной плоскости. Если бы было решено построить фазовый портрет, например, системы Рёсслера (A.17) или системы Лоренца (A.18), то необходимо было бы использовать трёхмерное пространство. Для динамических систем, описываемых ОДУ, их размерность равна числу уравнений первого порядка, на которые может быть разложена система. Для систем, описываемых отображениями последования, — числу скалярных переменных. Размерность систем, описываемых уравнениями с запаздыванием, в общем случае бесконечная, хотя и допускает часто хорошее конечномерное приближение. Если рассматривают не все динамические переменные (значение остальных фиксируют), говорят о «сечении» фазового пространства.

Применяя метод замены, уравнение ван дер Поля можно переписать следующим образом:

$$\begin{cases} \dfrac{dx_0}{dt} = x_1, \\ \dfrac{dx_1}{dt} = (r - x_0^2)x_1 - \omega_0 x_0. \end{cases}$$

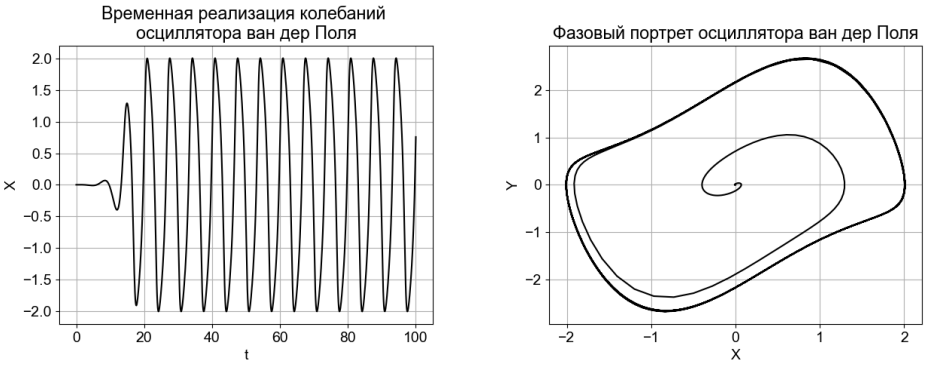
Решать эту систему дифференциальных уравнений первого порядка будем с помощью встроенной в модуль `scipy` функции `odeint`.

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

```
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

def func(x, t): # Пользовательская функция - оператор эволюции
    r = 1
    w = 1
    dx = np.zeros(2)
    dx[0] = x[1]
    dx[1] = (r-x[0]**2)*x[1] - (w**2)*x[0]
    return dx

# Начальные условия:
y0 = 0.001*np.ones(2)
# Моменты времени, в которые требуется получить решение:
t0 = np.linspace(0, 100, 1001)
X = odeint(func, y0, t0)
# Построение графиков:
plt.figure(1)
plt.plot(t0, X[:,0], color = 'black')
plt.grid()
plt.title('Временная реализация колебаний \nосциллятора ван дер Поля')
plt.xlabel('t')
plt.ylabel('X')
plt.show(1)
plt.figure(2)
plt.plot(X[:,0], X[:,1], color = 'black')
plt.grid()
plt.title('Фазовый портрет осциллятора ван дер Поля')
plt.xlabel('X')
plt.ylabel('Y')
plt.show(2)
```

Построим временную реализацию колебаний осциллятора ван дер Поля, см. рис. 12.8(a). Видно, что на отрезке времени $t \in [0, 100]$ система находится в разных состояниях: вначале наблюдается рост амплитуды колебаний, а затем выход на постоянное значение частоты и амплитуды. Состоянию $x(t)$ в некоторый момент времени t в фазовом пространстве будет соответствовать изображающая точка, характеризующая мгновенное состояние системы. В процессе эволюции изображающая точка с течением времени смещается вдоль некоторой линии — фазовой траектории. Совокупность характерных фазовых траекторий называют **фазовым портретом** системы.

Обратите внимание, что на рис. 12.8(b) можно выделить начальный участок траектории, стартующий из точки $(0, 0)$ — **переходной процесс**, и более поздний установившийся этап, характеризующийся большою степенью повторяемости, — установившийся режим движения. Установившиеся движения в фазовом пространстве называются **аттракторами** (образовано по всем правилам от лат. глагола attraho — притягиваю, соответственно, attractor — дословно притягатель, хотя в самой латыни такого слова не было). На рис. 12.8(b) аттрактором является **цикл** — замкнутая кривая, повторяющаяся с некоторым характерным периодом. Кроме цикла, существуют и другие виды



(a) (b)

Рис. 12.8. Временная реализация колебаний (a) и фазовый портрет (b) осциллятора ван дер Поля.

аттракторов. Самым простым аттрактором является **точка** — состояние равновесия. Ещё возможен **тор** — бесконечно тонкая нитка, наматывающаяся на бублик по кругу и никогда не попадающая в свой конец (не замыкающаяся) за счёт иррационального соотношения между диаметрами тора. Образом хаотических колебаний в фазовом пространстве является **странный хаотический аттрактор** – фрактальная линия бесконечной длины.

Для систем с непрерывным временем (потоков) действует следующее правило: при размерности фазового пространства $D = 1$ может существовать только аттрактор точка; при $D = 2$ — точки и циклы; при $D \geqslant 3$ — точки, циклы, торы и странные аттракторы. Как мы помним, осциллятор ван дер Поля описывается ОДУ, т. е. является системой с непрерывным временем, и обладает размерностью фазового пространства $D = 2$. Значит, в его фазовом пространстве возможен не только цикл, но и точка, которую можно получить при $r < 0$. Понимание того, что варианты установившихся движений и соответствующие им аттракторы ограничены размерностью динамической системы, на практике помогает определиться с выбором размерности математической модели. Например, наличие явно хаотического поведения говорит о том, что для моделирования такого объекта с помощью системы ОДУ первого порядка понадобится не менее трёх уравнений.

Для систем с дискретным временем (каскадов) складывается иная ситуация. Для них даже у системы первого порядка возможно хаотическое поведение, а у системы второго — квазипериодическое. Хотя каскадные системы могут быть рассмотрены как самостоятельные модели, на практике они часто получаются из потоковых — типичных для физики, химии и биологии моделей — путём упрощения: **сечения Пуанкаре**. В результате сечения аттрактора потоковой системы гиперплоскостью (в случае трёхмерной потоковой системы это обычная плоскость, а для двумерной системы — просто линия) образом аттрактора в виде точки или простого цикла становится точка, более сложных периодических аттракторов — несколько точек, тора — замкнутая кривая, а хаотического аттрактора — фрактал.

Множество точек в фазовом пространстве, из которых система попадает на аттрактор, называется **бассейном притяжения** данного аттрактора. При наличии нескольких аттракторов говорят о **мультистабильности**. Аттракторы могут существовать в пространстве состояний только **диссипативных** динамических систем. В физике так называются системы с трением (или другими потерями, например, из-за вязкости жидкости или нагрева резистора), в нелинейной динамике это более широкое понятие, так называют системы, обладающие свойством сжатия фазового объёма. В **консервативных** системах (в физике — системы без трения, в которых выполняется закон сохранения энергии) начальный фазовый объём сохраняется, лишь изменяя свою форму, следовательно, аттракторы в таких системах отсутствуют.

Чтобы закрепить и углубить полученные знания, построим также фазовый портрет трёхмерной системы Рёсслера (A.17) с помощью описанного выше метода Эйлера. При значениях параметров $a = b = 0.2$, $2.6 \leqslant c \leqslant 4.2$ система обладает устойчивым предельным циклом. Сразу же за точкой $c > 4.2$ возникает явление хаотического аттрактора.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import rcParams
rcParams['font.sans-serif']=['Liberation', 'Arial']
rcParams['font.size'] = 14.0

def Rossler(x, y, z):
    a = b = 0.2
    c = 5.7
    dx = -y-z
    dy = x+a*y
    dz = b+z*(x-c)
    return dx, dy, dz

dt = 0.01 #Шаг интегрирования
N = 10000

x0 = np.zeros(N+1)
y0 = np.zeros(N+1)
z0 = np.zeros(N+1)

x0[0] = 0.01; y0[0] = 0.02; z0[0] = 0.03 #Начальные условия

for i in range(N): #Метод Эйлера
    dx, dy, dz = Rossler(x0[i], y0[i], z0[i])
    x0[i+1] = x0[i] + (dx * dt)
    y0[i+1] = y0[i] + (dy * dt)
    z0[i+1] = z0[i] + (dz * dt)

fig = plt.figure()
ax = Axes3D(fig)
ax.plot(x0, y0, z0, color='black')
```

```
ax.set_title('Фазовый портрет системы Рёсслера')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

Рис. 12.9. Фазовый портрет системы Рёсслера.

12.3 Резонансные кривые

При гармоническом воздействии на линейный осциллятор $\frac{d^2x}{dt^2}+\omega_0^2 x = F\cos(\Omega t+\phi_0)$ наблюдается явление **линейного резонанса**: когда частота воздействия Ω приближается к собственной частоте осциллятора ω_0, амплитуда вынужденных колебаний A растёт. В осцилляторе без диссипации (написан выше) она стремится к бесконечности, в осцилляторе с потерями (А.2) амплитуда вынужденных колебаний достигает некоторого максимального значения, а затем уменьшается.

Зависимость амплитуды вынужденных колебаний от расстройки между частотой внешнего воздействия и собственной частотой осциллятора характеризуют **резонансные кривые**. Для линейного осциллятора под внешним гармоническим воздействием (А.2) соотношение между амплитудою внешних колебаний и частотою внешнего воздействия определяется формулою (12.12).

$$A(\Omega) = \frac{F}{\sqrt{\left(\omega_0^2 - \Omega^2\right)^2 + 4\gamma^2\Omega^2}} \tag{12.12}$$

Построим график (рис. 12.10) с резонансными кривыми для диссипативного линейного осциллятора под гармоническим воздействием. Зафиксируем собственную частоту колебаний линейного осциллятора $\omega_0 = 1$ и амплитуду внешнего воздействия $F = 1$. Частоту внешнего воздействия будем менять в диапазоне $\Omega \in [0, 2]$ с шагом 0.01. Графики построим для значений коэффициента затухания $\gamma = 0, 0.1, 0.2, 0.3$.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
rcParams['font.size'] = 14.0

w0 = 1 # Собственная частота колебаний осциллятора
F = 1 # Амплитуда внешнего воздействия

def OnePlot(gamma):
    """ Функция для построения резонансной кривой при
        одном из значений gamma """
    # Словарь зависимости положения конца указательной стрелки
    от значения gamma:
    xy = {0: (1.075, 6), 0.1: (1.055,4), 0.2: (1.015,2.4), 0.3: (0.95,1.8)}
    # Словарь зависимости положения текста на графике от значения gamma:
    xyt = {0: (1.6, 7), 0.1: (1.6, 5.5), 0.2: (1.6, 4), 0.3: (1.6, 2.5)}
    # Возможные значения частоты внешнего воздействия:
    Omega = np.arange(0,2,0.01)
    A = []
    for o in Omega:
        a = F/np.sqrt((w0**2-o**2)**2 + 4*gamma**2*o**2)
        A.append(a)
    plt.plot(Omega, A, color = 'grey')
    plt.annotate(r'$\gamma = $' +str(gamma), xy[gamma], xytext=xyt[gamma],
                 arrowprops=dict(facecolor='black', arrowstyle = '->'))

gamma = np.arange(0,0.4,0.1)
for g in gamma:
    OnePlot(round(g,2))

plt.title('Резонансные кривые линейного осциллятора', fontsize = 12.0)
plt.text(0.25, 8, r'$\omega_0 = 1$'+'\n'+ r'$F = 1$')
plt.ylim(0,10)
plt.xlabel(r'$\Omega$')
plt.ylabel('A')
plt.show()
```

В данном примере использованы две новые для нас функции: `text` и `annotate` из модуля `matplotlib.pyplot`, которые позволяют создавать текстовые элементы на графиках. С помощью `text` на графике указаны значения ω_0 и F. С помощью `annotate`

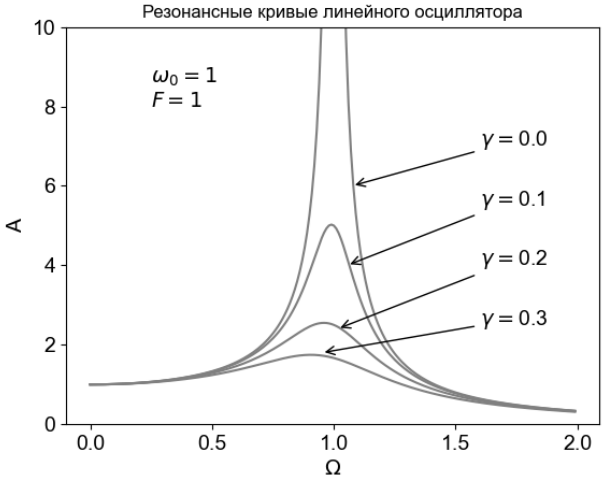


Рис. 12.10. Резонансные кривые линейного осциллятора.

были сделаны надписи $\gamma = ...$ и нарисованы стрелочки к соответствующим резонансным кривым.

Функция `text (x, y, text)` отвечает за установку текстовых блоков на поле графика. У этой функции три основных параметра, с помощью которых можно задать расположение и содержание: `x` — значение левого нижнего края надписи по оси x (тип `float`), `y` — значение левого нижнего края надписи по оси y (тип `float`), `text` — текст надписи (тип `str`). Дополнительные настройки шрифта осуществляются именованными параметрами, например, можно дописать `fontsize=14`.

Функция `annotate()` позволяет установить текстовый блок с заданным содержанием и стрелкой для конкретного места на графике. Аннотация — это мощный инструмент, подробнее о нём при необходимости можно почитать в официальном руководстве на сайте `https://matplotlib.org`. В нашем примере `annotate ('text', (x, y), xytext, arrowprops)` были использованы два обязательных параметра и два именованных параметра: `text` — текст надписи (тип `str`), `(x, y)` — координаты места, на которое будет указывать стрелка (кортеж), `xytext` — нижний левый край надписи и начало стрелки (кортеж) и `arrowprops` — параметры отображения стрелки (у нас только цвет и стиль, на самом деле, настроек у этого параметра гораздо больше).

А теперь вновь окунёмся в теорию колебаний. Наиболее важным является поведение кривой (12.12) при значениях частоты внешнего воздействия, близких к частоте собственных колебаний. В таком случае вводят частотную расстройку $\delta = (\Omega - \omega_0)$, через которую выражение для резонансной кривой (12.12) переписывается следующим

образом (приблизительное равенство справедливо при $\delta << \omega_0$):

$$A(\Omega) = \frac{F}{\sqrt{\delta^2 \left(2\omega_0 - \delta\right)^2 + 4\gamma^2(\omega_0 - \delta)^2}}, \quad A(\delta)\sqrt{\gamma^2 + \delta^2} \approx \frac{F}{2\omega_0} \tag{12.13}$$

В нелинейном осцилляторе при гармоническом воздействии наблюдается явление **нелинейного резонанса**. Точное решение для нелинейного осциллятора получить нельзя, но можно получить приближённое. Для этого существуют несколько методов. Для диссипативного осциллятора Дуффинга (А.4), как правило, используется метод медленно меняющихся комплексных амплитуд (см., например, книгу «А. П. Кузнецов С.П. Кузнецов, Н.М. Рыскин. Нелинейные Колебания: Учеб. пособие для вузов. — М.: Изд-во физико-математической литературы, 2002. — 292 с.», далее Кузнецов и др.). Таким образом можно, в том числе, получить приближённое выражение для связи амплитуды колебаний и расстройки (12.14).

$$A\sqrt{\gamma^2 + \left(\delta - \frac{3\beta A^2}{8\omega_0}\right)^2} \approx \frac{F}{2\omega_0}, \tag{12.14}$$

где величины A, δ и F, ω_0, γ значат то же, что и для линейного осциллятора, а величина β есть коэффициент нелинейности — см. (А.4). Выражение (12.14) в отличии от (12.13) может давать от одного до трёх решений для A при одном и том же δ. Нетрудно видеть, что выражение для нелинейного резонанса (12.14) переходит в (12.13) при $\beta = 0$.

Далее вводятся следующие обозначения: $P = (3\beta F^2)/(32\gamma^3\omega_0^3)$ — безразмерный параметр, характеризующий интенсивность внешнего воздействия; $X = (3\beta A^2)/(8\gamma\omega_0)$ — безразмерный параметр, характеризующий интенсивность вынужденных колебаний, $\Delta = \delta/\gamma$ — безразмерная расстройка частоты. Тогда соотношение (12.14) примет вид:

$$X = \frac{P}{(X - \Delta)^2 + 1}. \tag{12.15}$$

Построить кривую $X(\Delta)$ по формуле (12.15) оказывается не так-то просто. До этого мы всегда строили графики однозначных функций, у которых одному значению аргумента (в нашем случае Δ) соответствует одно значение функции (в нашем случае X). Формула (12.15) на самом деле может быть представлена как кубическое уравнение (12.15) относительно X, где Δ — просто коэффициент. Таким образом, одному и тому же Δ может соответствовать от 1 до 3 разных X. Такая функция называется неоднозначной (многозначной).

$$X((X - \Delta)^2 + 1) - P = 0. \tag{12.16}$$

Существуют два пути построения необходимой кривой. Либо нужно решить уравнение (12.16) численно, например, методом Ньютона, но тогда нужно проводить решение для каждого Δ минимум трижды с разными начальными догадками (или на разных отрезках), чтобы найти все возможные корни. Либо решить его аналитически. Средствами Python возможно и то и другое.

Численные методы работают с математическими равенствами и формулами как с последовательностью чисел. Занимаются разработкой и реализацией решения математических задач в численном виде на компьютере.

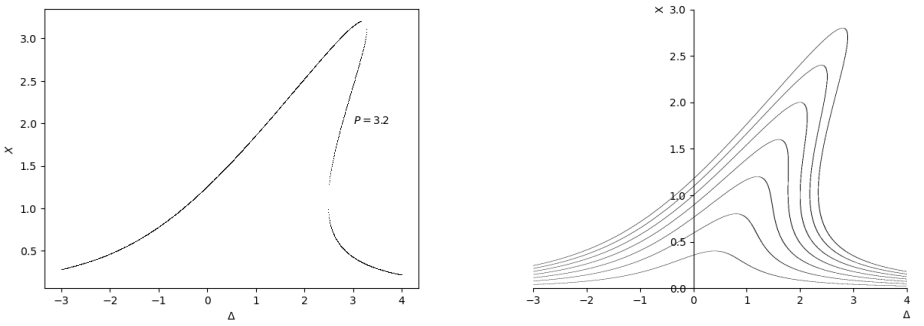
Символьные вычисления работают с математическими равенствами и формулами как с последовательностью символов. Занимаются разработкой и реализацией аналитических методов решения математических задач на компьютере.

Сначала рассмотрим численное решение средствами функции `root` из модуля `scipy.optimize`. Функция принимает на вход другую функцию — уравнение, которое нужно решить, а также начальную догадку. Кроме того, через кортеж `args` мы передали в неё Δ и P и метод, которым будет проводиться решение — в примере использован метод Крылова, являющийся обобщением метода Ньютона. Для каждого Δ вызов осуществляется трижды с начальными значениями 1, 2 и 3 соответственно, чтобы найти все три возможных решения. В действительности, это не гарантировано, разные начальные условия могут привести к одному и тому же решению и запускать лучше больше трёх раз, но в приведённом примере такой подход работает. Если в листинге поменять цвета, которым строятся массив `XX2` и `XX3`, можно видеть, какая часть графика построена из каких начальных условий. График с резонансными кривыми диссипативного нелинейного осциллятора (осциллятора Дуффинга) будем строить при фиксированном значении параметра интенсивности внешнего воздействия $P = 2.8$, безразмерную расстройку частоты будем плавно менять в диапазоне $\Delta \in [-3, 4]$ с шагом 0.005.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root

P = 3.2
Deltas=np.arange(-3, 4, 0.005)

def resCurveDuff(X, Delta, F):
    return X * (1 + (Delta - X)**2) - P

def poliDelta(X0):
    XX = []
    for Delta in Deltas:
        try:
            res = root(resCurveDuff, X0, args=(Delta, P), method='krylov')
        except ValueError:
            print(Delta, X0)
        if res.success:
            if res.x > 0:
                XX.append([Delta, res.x])
    return np.array(XX).T

XX1 = poliDelta(1.)
XX2 = poliDelta(2.)
XX3 = poliDelta(3.)

plt.figure(figsize=(16./2.54, 12./2.54))
plt.plot(XX1[0], XX1[1], ',', color='black')
plt.plot(XX2[0], XX2[1], ',', color='black')
plt.plot(XX3[0], XX3[1], ',', color='black')
plt.xlabel(r'$\Delta$')
plt.ylabel(r'$X$')
plt.text(3, 2.0, r'$P='+str(P)+r'$')
```

```
plt.show()
```

Результатом вызова функции `root` является объект, в поле `.x` которого содержится решение. Функция `root` не всегда справляется с поставленною задачею, иногда метод расходится. В этом случае поле `.success` логического типа будет содержать ложь; такие значения мы не будем использовать, как видно по листингу. К сожалению, изредка сбой происходит не в Python-функции `root`, а в вызываемой ею процедуре, написанной обычно на Фортране. В таком случае программа аварийно завершается с ошибкою `ValueError`, поскольку происходит недопустимое действие, например, берётся корень из отрицательного числа или происходит деление на ноль, причём не обязательно в самой функции, а часто в её производной, обязательно используемой в ньютоновских методах. Чтобы исключить аварийное завершение программы по этой причине, мы использовали блок **`try`**, **`except`** и при ошибке просто выводили на экран значения Δ и X, при которых возникает сбой.



(a) (b)

Рис. 12.11. Резонансные кривые для осциллятора Дуффинга, построенные численно с использованием функции `root` из `scipy.optimize` — a, и аналитически с помощью пакета `sympy` — b.

Получившийся результат можно видеть на рис. 12.11(a). Видно, что удалось выявить неоднозначное поведение кривой, хотя и не идеальным образом. В физической системе из-за такой неоднозначности будет наблюдаться явление **гистерезиса**: при проходе слева-направо амплитуда будет расти, а потом внезапно сорвётся почти возле максимума — движение происходит по верхней ветке резонансной кривой (подробнее см. Кузнецов и др.). В обратном направлении справа-налево амплитуда будет медленно расти и при Δ, при которых наблюдаются максимальные значения, всё ещё будет мала и лишь позднее перепрыгнет на большие значения, но максимум уже не будет достигнут.

Решить уравнение (12.16) аналитически можно, используя пакет `sympy`, предназначенный для символьных вычислений. Значение параметра интенсивности внешнего воздействия будем менять в диапазоне $P \in [0.4, 2.8]$ с шагом 0.2. Продемонстрированное в следующем листинге и нарисованное на рис. 12.11(b) решение совершеннее и получено гораздо быстрее (когда вы реализуете программу, описанную выше, то

можете заметить, что считается она небыстро), чем численное решение средствами `scipy.optimize.root` (построение графика осуществляется собственными средствами пакета `sympy`).

Итак, из модуля `sympy` нам понадобятся три функции: `symbols()`, `Eq()` и `plot_implicit()`.

```
import numpy as np
from sympy import symbols, Eq, plot_implicit
D, X = symbols(r'$\Delta$ X') # Объявление переменных символьными
Plot1 = plot_implicit(Eq(X+X*(D-X)**2, 2.8), (D, -3, 4), (X, 0, 3),
        line_color='black', show = False)
for F in np.arange(0.4, 2.8, 0.4):
    Plot2 = plot_implicit(Eq(X+X*(D-X)**2, F), (D, -3, 4), (X, 0, 3),
            line_color='black', show = False)
    Plot1.extend(Plot2)
Plot1.show()
```

Функция `symbols()` необходима для того, чтобы принудительно объявить наши переменные символьными. В языках, специально предназначенных для символьных вычислений, таких, как Mathematica или Maple, если используется переменная, которой ничего не было присвоено, то она автоматически воспринимается как символ с тем же именем. Python не был изначально предназначен для символьных вычислений, поэтому при использовании переменной, которой ничего не было присвоено, вы получите сообщение об ошибке. Объекты типа `Symbol` нужно создавать явно. Имя символа и имя переменной, которой мы присваиваем этот символ — две независимые вещи, и можно написать `a, b, c = Symbol('x␣y␣z')`. Но тогда при вводе программы вы будете использовать переменные `a, b, c`, а при печати результатов `sympy` будет использовать `x y z`.

Функция `Eq()` позволят задавать уравнения, имеет два обязательных параметра: левую и правую части решаемого уравнения.

Функция `plot_implicit()` позволяет строить графики неявных функций, имеет три обязательных параметра (уравнение, диапазоны по осям) и много именованных параметров, из которых мы использовали два (прописали цвет линии и попросили не показывать график на каждом шаге цикла).

12.4 Расчёт старшего ляпуновского показателя

Наиболее популярной характеристикой для описания поведения нелинейных систем является ляпуновский показатель. Ляпуновский показатель характеризует поведение двух изначально очень близких точек в фазовом пространстве. Малое отклонение изображающей точки от некоторой траектории на аттракторе, то есть малое возмущение ε_0, до тех пор, пока оно не достигло значительных величин, эволюционирует приближённо по экспоненциальному закону вида $\varepsilon(t) = \varepsilon_0 e^{\lambda \Delta t}$. В результате D-мерная (где D — число уравнений первого порядка, на которые может быть разложена система) сфера множества начальных возмущений через некоторое время трансформируется в эллипсоид. Измерив эволюцию возмущения через время τ, можно вычислить показатели экспоненты, как отношения длин полуосей эллипса возмущений к начальному радиусу: $\lambda_i = \frac{1}{\tau} ln \frac{\varepsilon_i}{\varepsilon_0}$. Усреднённые по всему аттрактору значения этих коэффициентов

называют **показателями Ляпунова**, мы их обозначим $\Lambda_1, \Lambda_2, ..., \Lambda_D$. Они характеризуют устойчивость движения на аттракторе в линейном приближении. Упорядоченный по убыванию набор значений Λ_i называют **спектром ляпуновских показателей**, а последовательная запись их знаков (+, – или 0) — сигнатурой спектра. Если все показатели отрицательны, то есть сигнатура имеет вид (–, –,..., –), то аттрактором является точка равновесия. Сигнатура предельного цикла — (0, –,..., –), для двумерного тора — (0, 0, –, ..., –). В спектре ляпуновских показателей хаотического аттрактора обязательно присутствует хотя бы один положительный показатель, определяющий разбегание изначально близких траекторий, например, (+, 0, –,..., –); при этом сумма всех ляпуновских показателей для любого аттрактора должна быть отрицательна, иначе траектория глобально (на больших временах) теряет устойчивость.

На практике проще всего посчитать старший ляпуновский показатель Λ_1, а не весь спектр, поскольку для этого не нужно вводить сложные перенормировки в фазовом пространстве (процесс Грама–Шмидта) и вычислительно это не столь трудоёмкий процесс. К тому же именно старший показатель может больше всего сказать о режиме в системе.

Расчёт старшего показателя Ляпунова продемонстрируем на примере логистического отображения (А.33). Логистическое отображение (также квадратичное отображение или отображение Фейгенбаума) — это полиномиальное отображение, которое описывает, как меняется численность популяции с течением времени. Его часто приводят в пример того, как из очень простых нелинейных уравнений может возникать сложное, хаотическое поведение. Математическая формулировка отображения:

$$x_{n+1} \quad = \quad rx_n(1 - x_n),$$

где x_n принимает значения от 0 до 1 и отражает численность популяции в n-ом году, а x_0 обозначает начальную численность (в год номер 0); r — положительный параметр, характеризующий скорость размножения (роста) популяции.

При изменении значения параметра r в системе наблюдается следующее поведение. При $r \leqslant 2$ численность популяции сокращается до нуля. Если $2 < r \leqslant 3$, численность популяции придёт к стационарному ненулевому значению $(r - 1)/r$, но вначале будет несколько колебаться вокруг него. Если $3 < r \leqslant 3.45$, численность популяции будет бесконечно колебаться между двумя значениями. Если $3.45 < r \leqslant 3.54$, то численность популяции будет бесконечно колебаться между четырьмя значениями. При дальнейшем увеличении $r > 3.54$, численность популяции будет колебаться между 8 значениями, потом 16, 32 и так далее. Длина интервала изменения параметра, при котором наблюдаются колебания между одинаковым количеством значений, уменьшается по мере увеличения r — режимам поведения со всё большим числом различных последовательно сменяемых состояний соответствуют всё меньшие диапазоны параметра. Подобное поведение является типичным примером каскада **бифуркаций** удвоения периода (об этом явлении мы поговорим чуть ниже). При значении $r \approx 3.57$, каскад удвоений заканчивается и начинается хаотическое поведение — колебания становятся нерегулярными (каждое последовательное значение уникально). Небольшие изменения в начальных условиях приводят к несопоставимым отличиям дальнейшего поведения системы на больших промежутках времени, например, можно изменить x на 10^{-6} и через некоторое число шагов получить различия в значениях x порядка 1, что является основною характеристикой хаотического поведения.

Большинство значений r, превышающих 3.57, соответствуют хаотическому поведению, однако существуют узкие, изолированные «окна» значений r, при которых систе-

ма ведет себя регулярно, называемые «окнами периодичности». К примеру, начиная со значения приблизительно 3.83, существует интервал параметров, при котором наблюдаются колебания между тремя значениями, а для чуть больших значений r — между 6, потом 12 и т. д. Фактически, в системе можно найти периодические колебания с любым количеством значений. При $r > 4$, значения отображения покидают интервал $[0, 1]$ и **расходятся** (устремляются к бесконечности) при любых начальных условиях — аттрактор разрушается.

Графическое изображение вышеперечисленного можно продемонстрировать с помощью графика зависимости старшего ляпуновского показателя от параметра r. По оси абсцисс отложим значения параметра r, а по оси ординат — полученные значения старшего показателя Ляпунова Λ_1.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']

epslyap = 1e-6
r = np.arange(2.4, 4.0, 0.001)
N = 50
Lyap = [] #Список значений старшего показателя Ляпунова
for j in range(len(r)):
    #Пропускаем переходной процесс
    x = 0.1
    for i in range(10000):
        x = r[j]*x*(1-x)
    #Вычисляем исходный x и возмущённый x
    X = x
    X_perturbation = x + epslyap

    LyapLoc = [] #Список локальных показателей Ляпунова
    for i in range(N-1):
        X = r[j]*X*(1-X)
        X_perturbation = r[j]*X_perturbation*(1-X_perturbation)
        delta = abs(X_perturbation - X)
        LyapLoc.append(np.log(delta/epslyap))
        X_perturbation = X + (X_perturbation - X) * (epslyap/delta)
    Lyap.append(np.mean(LyapLoc))

plt.plot(r, Lyap, '-', color = 'black')
plt.xlim(2.4, 4.0)
plt.title('Старший ляпуновский показатель')
plt.xlabel('r')
plt.ylabel(r'$\Lambda_1$')
plt.grid()
plt.show()
```

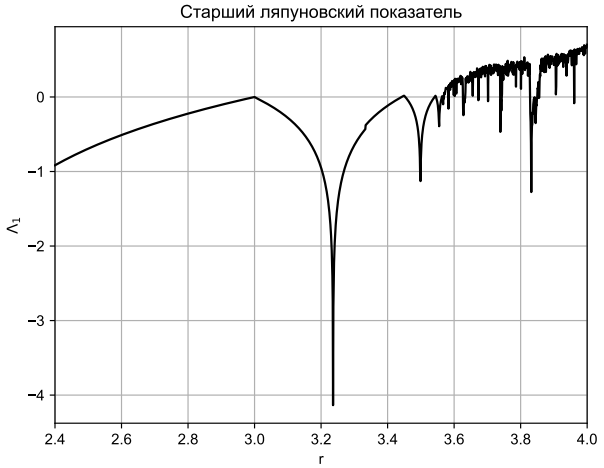


Рис. 12.12. Зависимость значения старшего ляпуновского показателя Λ_1 от параметра r для логистического отображения.

По графику зависимости старшего ляпуновского показателя (рис. 12.12) легко определять значения параметра, когда происходят бифуркации удвоения периода. В эти моменты старший показатель Ляпунова становится равен нулю $\Lambda_1 = 0$ (с точностью до погрешности вычисления). Там, где $\Lambda_1 > 0$, в системе наблюдается хаотическое поведение. Окно периодичности также хорошо видно на этом графике, оно наблюдается, когда $\Lambda_1 < 0$ при $r \approx 3.83$.

Напоминаем, что логистическое отображение является каскадной одномерной системой. Теперь построим зависимость значения старшего ляпуновского показателя для потоковой трёхмерной системы — модели системы фазовой автоподстройки частоты (A.23).

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
from scipy.integrate import solve_ivp

epslyap = 1e-6
eps1=9.0
eps2=10.0
T = 1.

def Neuron(t, x, gamma):
    dx = np.zeros(3)
```

```
    dx[0] = x[1]
    dx[1] = x[2]
    dx[2] = gamma/(eps1*eps2) - (eps1+eps2)/(eps1*eps2)*x[2] \
            - (1+eps1*np.cos(x[0]))/(eps1*eps2)*x[1]
    return dx

gammas = np.arange(0.025, 0.31, 0.025)
N = 2000
Ttrans = 10000
Lyap = [] #Список значений старшего показателя Ляпунова
for gamma in gammas:
    # Пропускаем переходной процесс
    x0 = [0.1, 0.1, 0.1]
    solution = solve_ivp(Neuron, (0, Ttrans), x0, args=(gamma,)) #для потоков
    # Формируем исходный x и возмущённый x
    X = solution.y[:, -1]
    X_perturbation = np.copy(X)
    X_perturbation[1] += epslyap  # возмущение вносим в y
    LyapLoc = [] # список локальных показателей Ляпунова
    for i in range(N):
        # Решаем невозмущённую систему на интервале T:
        solution = solve_ivp(Neuron, (0, T), X, args=(gamma,))
        Xnov = solution.y[:, -1]
        # Решаем возмущённую систему на том же интервале:
        solutionp = solve_ivp(Neuron, (0, T), X_perturbation, args=(gamma,))
        Xnovp = solutionp.y[:, -1]
        # Вычисляем изменение возмущения:
        delta = np.sqrt(np.sum((Xnovp - Xnov)**2))
        LyapLoc.append(np.log(delta/epslyap))
        # Перенормируем возмущение (делается для многомерных систем):
        X_perturbation = Xnov + (Xnovp - Xnov) * (epslyap/delta)
        X = Xnov
    Lyap.append(np.mean(LyapLoc))

plt.plot(gammas, Lyap, '-', color = 'black')
plt.xlim(0, 0.3)
plt.title('Старший ляпуновский показатель')
plt.xlabel(r'$\gamma$')
plt.ylabel(r'$\Lambda_1$')
plt.grid()
plt.show()
```

Интегрирование трёхмерной потоковой системы гораздо более трудоёмкий процесс, чем итерирование одномерного отображения, поэтому график 12.13 получился совсем не такой гладкия и ясный, как 12.12. Следует отметить, что рассмотренная система (A.23) при ненулевом параметре γ лишена состояния равновесия и всегода находится в колебательном режиме, поэтому малые значения $\Lambda_1 < 0.0025$ на самом деле соответствуют нулевым значениям и являются результатом численной погрешности расчётов,

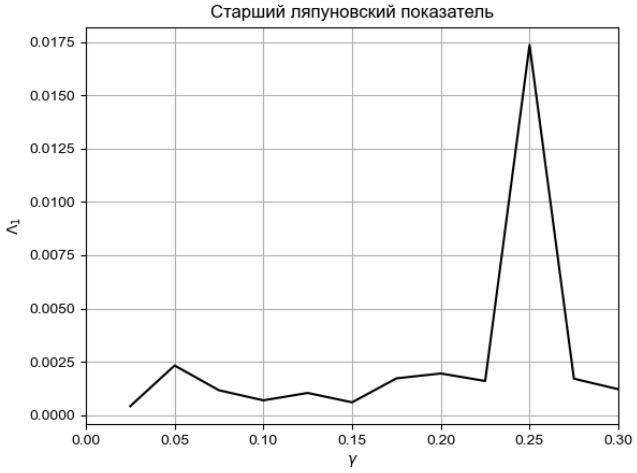


Рис. 12.13. Зависимость значения старшего ляпуновского показателя Λ_1 от параметра γ для модели системы фазовой автоподстройки частоты.

которую можно снизить путём увеличения длины переходного процесса $Ttrans$ и числа N усредняемых локальных ляпуновских показателей. При $\gamma = 0.25$ наблюдается резкий выброс, соответствующий хаотическому режиму.

12.5 Бифуркационные диаграммы

Ещё одним популярным вариантом описания поведения нелинейных систем являются бифуркационные диаграммы. **Бифуркация** — это качественное изменение поведения динамической системы при бесконечно малом изменении её параметров. Следует сразу отметить, что плавная деформация аттрактора и сопровождающие её изменения формы колебаний не считаются качественными изменениями. Параметр, изменение которого приводит к бифуркации, называется **бифуркационным параметром**. Под понятием **бифуркационная диаграмма** подразумевают изображение на рисунке смены возможных динамических режимов системы (равновесных состояний, периодических орбит, квазипериодических орбит и хаотического поведения) при изменении значения бифуркационного параметра. Бифуркационная диаграмма — это пример построения комбинированного пространства, где по оси абсцисс откладывается значения бифуркационного параметра, а по оси ординат — значения динамической переменной в установившемся режиме. Построение бифуркационной диаграммы рассмотрим также на примере логистического отображения.

```
import numpy as np
import matplotlib.pyplot as plt
r = np.arange(2.4, 4.0, 0.001)
```

```
N = 50
Ntrans = 10000
Mas = np.zeros ((len(r),N))
for j in range(len(r)):
    x = 0.1
    for i in range(Ntrans):
        x = r[j]*x*(1-x)
    X = np.zeros(N)
    X[0] = x
    Mas[j,0] = X[0]
    for i in range(N-1):
        X[i+1] = r[j]*X[i]*(1-X[i])
        Mas[j,i+1] = X[i+1]
plt.plot(r, Mas, ',', color = 'black')
plt.xlim(2.4, 4.0)
plt.title('Бифуркационная диаграмма')
plt.xlabel('r')
plt.ylabel('x')
plt.show()
```

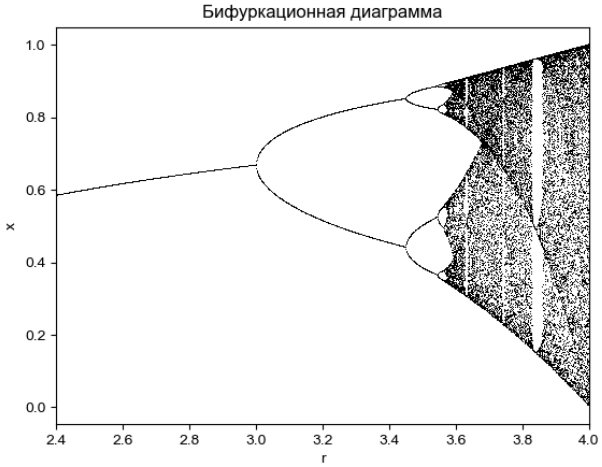


Рис. 12.14. Бифуркационная диаграмма для логистического отображения.

На практике бифуркационные диаграммы строят, чтобы определить диапазоны параметра, в которых существуют желаемые и наоборот нежелаемые режимы. Например, для модели типа хищник–жертва такой подход позволяет понять, при каких значениях параметров воспроизводства обе популяции будут существовать, а при каких — вымрут. Для систем фазовой автоподстройки частоты — узнать, насколько можно менять

параметры, чтобы остаться в режиме устойчивой периодической генерации. Для систем связи на хаотической несущей (см. «Генерация хаоса / Под общ. ред. Дмитриева А.С. Москва: Техносфера, 2012. — 424 с.») — напротив выявить значения параметров, пригодные для широкополосной передачи сигнала.

12.6 Карта режимов (пространство параметров)

Выше мы уже подробно разобрали такие понятия, как пространство состояний (фазовый портрет) и комбинированное пространство (бифуркационная диаграмма), осталось ещё одно понятие — **пространство параметров**, когда по осям откладываются значения параметров. С помощью такой визуализации можно наглядно показать всю совокупность видов установившихся движений и переходов между ними, возможных в рассматриваемой динамической системе. **Карта динамических режимов** — диаграмма на плоскости параметров, где области различных режимов динамики показаны определенным цветом. Простейший способ построения карты состоит в том, чтобы просканировать шаг за шагом всю интересующую область. При этом в каждой точке, отвечающей одному пикселю, проводится решение дифференциального уравнения (или итерируется отображение для систем с дискретным временем), затем анализируется характер режима, возникающего после завершения переходного процесса и выхода на аттрактор, и точка отмечается соответствующим цветом. В тех случаях, когда имеет место мультистабильность, т.е. сосуществует два и более аттракторов при одних и тех же параметрах, карту динамических режимов надо представлять как совокупность перекрывающихся листов со своей раскраской на каждом из них.

Пояснение алгоритма построения карт динамических режимов лучше начинать с каскадов. Построим карту режимов для отображения Эно (А.36), у которого два параметра α и β. Параметр α будем менять в диапазоне $\alpha \in [0, 2.0]$ с шагом 0.01. Параметр β будем менять в диапазоне $\beta \in [-0.5, 0.5]$ с шагом 0.005.

$$\begin{cases} x_{n+1} = 1 - \alpha x_n^2 - \beta y_n, \\ y_{n+1} = x_n. \end{cases}$$

Для визуализации будем использовать функцию `imshow` из подмодуля `pyplot` модуля `matplotlib`. Эта функция отображает цветами на экране двумерный массив. При некоторых наборах параметров отображение Эно может «разбегаться» — уходит на бесконечность. Чтобы не допускать такие ситуации, в программе использовался стандартный приём «флаг», который в такой случае позволял обозначить этот режим за ноль и обеспечивает выход из цикла по `break`.

Внимание! Программа может считать полчаса и более, не пугайтесь.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial', 'Dejavu Sans']
rcParams['font.size']=24

delta = 0.01
maxregim = 13 # максимально разрешимый режим
Ntrans = 10000
```

```
def Henon(x, alpha, beta):
dx = np.zeros(2)
dx[0] = 1 - alpha*x[0]**2- beta*x[1]
dx[1] = x[0]
return dx

plt.figure(figsize = (9, 9))
a_min = 0; a_d = 0.01; a_max = 2
b_min = -0.5; b_d = 0.005; b_max = 0.5
a_n = np.arange(a_min, a_max+delta, a_d)
b_n = np.arange (b_min, b_max+delta, b_d)
map_Henon = np.zeros((len(a_n), len(b_n)), dtype=int)
for i, a in enumerate(a_n):
    for j, b in enumerate(b_n):
        #Пропускаем переходной процесс длины Ntrans
        x = [0.1, 0.1] #начальные условия
        flag = False
        for n in range(Ntrans):
            x = Henon (x, a, b)
            if abs(x[0])>10:
                flag = True
                break
        if not flag:
            y_sec = [x[0]]
            for n in range(maxregim+1):
                x = Henon (x, a, b)
                y_sec.append(x[0])
            i_y = 1
            while i_y < len(y_sec) and abs(y_sec[i_y]-y_sec[0])>delta:
                i_y = i_y + 1
            if i_y > maxregim: i_y = maxregim
            map_Henon[i,j] = i_y

plt.imshow(map_Henon, cmap = 'jet', origin='lower',
           extent = (b_min, b_max, a_min, a_max), aspect = b_d / a_d)
plt.colorbar(boundaries=np.arange(-0.5, maxregim+1.5, 1),
             ticks=range(0, maxregim+1), fraction=0.044)
plt.xlabel(r'$\beta$', fontsize=28)
plt.ylabel(r'$\alpha$', fontsize=28)
plt.tight_layout()
plt.savefig('map_Henon_1thread.png')
plt.show()
```

В итоге получим вот такую (рис. 12.15) карту режимов отображения Эно на плоскости параметров (α, β). Периодические движения с периодом от 1 до 12 представлены цветами от тёмно-синего до тёмно-красного. Хорошо видны линии бифуркации

удвоения периода: от 1 к 2, от 2 к 4, от 4 к 8. Бордовым (цвет 13) обозначены как периодические движения большего периода, так и хаотические режимы.

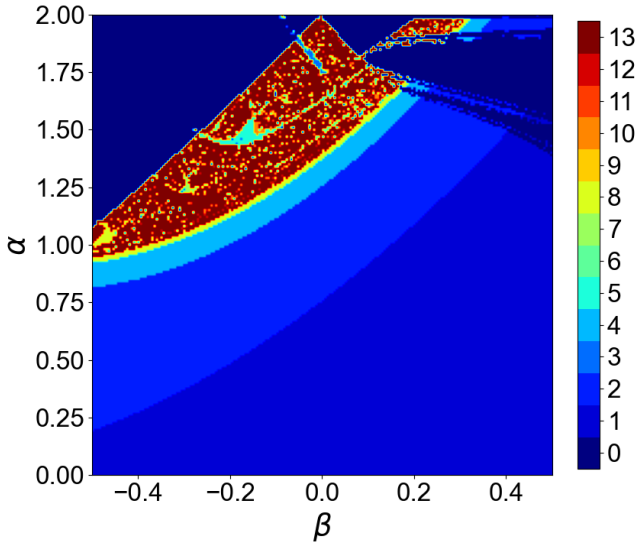


Рис. 12.15. Карта режимов отображения Эно на плоскости параметров (α, β). Периодические движения с периодом от 1 до 12 представлены цветами от тёмно-синего до тёмно-красного. Бордовым (цвет 13) обозначены как периодические движения большего периода, так и хаотические режимы.

Теперь построим карту режимов потоковой системы — системы Рёсслера (A.17). В случае потоков необходимо использовать сечение (отображение) Пуанкаре. Французский математик Анри Пуанкаре предложил ставший классическим метод анализа динамических систем. Этот метод позволяет заменить потоковую систему n-го порядка на отображение $(n-1)$-го порядка с дискретным временем, называемое отображением Пуанкаре. В случае трёхмерной системы, каковой является система Рёсслера, сечение нужно делать плоскостью, пересекающей траектории, причём так, чтобы избежать касаний, например, можно сделать сечение плоскостью $x = 0$. Фактически это означает, что необходимо во временном ряде $\{(x_n, y_n, z_n)\}_{n=1}^{N}$, полученном, например, решением системы Рёсслера методом Рунге–Кутты на достаточно длинном временном отрезке $T = N\Delta t$, где Δt — интервал дискретизации, N — число точек в ряде, отыскивать значения координат (y, z) (на практике ограничиваются только одною координатой, чаще всего y), соответствующие прохождению траекторией значения $x = 0$, причём всегда в одном и том же направлении: либо увеличения x (значит, во временном ряде $x_n < 0$,

а $x_{n+1} \geqslant 0$), либо уменьшения (во временном ряде $x_n > 0$, а $x_{n+1} \leqslant 0$). В простейшем случае в качестве значений отображения Пуанкаре можно брать y_n, удовлетворяющие условию $x_n < 0$, а $x_{n+1} \geqslant 0$, но такой подход требует очень малого шага Δt для более-менее точного определения режима поведения, поскольку вместо точки пересечения плоскости берётся следующая за нею точка по траектории. Точность можно существенно повысить, если аппроксимировать точку пересечения, используя линейную аппроксимацию — уравнение прямой, проходящей через две точки на плоскости:

$$\frac{y - y_n}{y_{n+1} - y_n} = \frac{x - x_n}{x_{n+1} - x_n} \tag{12.17}$$

Если в (12.17) подставить $x = 0$ и выразить оттуда y, то полученное значение y можно рассматривать как очередное значение отображения Пуанкаре.

Поскольку заранее неизвестно, сколько точек придётся на один виток кривой и, следовательно, сколько пересечений плоскости удастся детектировать на длине ряда в N значений, величину N обычно приходится подбирать эмпирически так, чтобы минимальное число пересечений было больше, чем самый сложный из распознаваемых режимов: например, если хочется распознать режим периода 12 (в нашей программе переменная, отвечающая за этот параметр, будет называться `maxregime`), когда траектория попадает в свой конец через 16 витков, нужно иметь не менее 13 пересечений.

Расчёт для потоковых систем, как правило, гораздо более вычислительно затратный, чем для отображений, так как кроме необходимости решения системы дифференциальных уравнений вместо простого итерирования отображения, для одного шага сечения Пуанкере часто оказывается нужно сделать десятки или сотни шагов алгоритма, ведь чем меньше Δt и, следовательно, больше шагов на одном витке, тем точнее будет детектировано значение y, при котором $x = 0$ и меньше ошибок в определении режимов будет допущено. Однако поскольку для каждого набора параметров расчёт можно производить независимо, проблема может быть эффективно решена с использованием многоядерных компьютеров и видеоускорителей.

Проблему распараллеливания программы мы будем решать в другой главе, а пока попробуем написать простую программу, строящую карту режимов системы Рёсслера, пусть и не с самых хорошим разрешением. Для построения карты режимов системы Рёсслера будем параметр a менять в пределах $[0.1; 0.34]$ с шагом 0.003 (это недостаточно маленький шаг, но для учебных целей подойдёт), параметр c будем менять в пределах $[1; 5]$ с шагом 0.05. Решать систему ОДУ будем встроенной функцией `solve_ivp`, которая лежит в `scipy.integrate`, метод выставим `LSODA` — это обёртка над библиотекою `LSODA`, написанной на Фортране, что обеспечивает высокую сравнительно с другими методами скорость вычислений при приемлемой точности. `LSODA` использует автоматический подбор шага интегрирования, который может отличаться от шага дискретизации (в нашей программе обозначается `dt`) в меньшую сторону, если это необходимо для достижения нужной точности вычисления.

```
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial', 'Dejavu Sans']
rcParams['font.size']=24
```

```
b = 0.2
delta = 0.01
dt = 0.001
N = 2000
maxregime = 13 # максимально разрешённый режим

def Rossler(t, x):
    dx = np.zeros(3)
    dx[0] = -x[1]-x[2]
    dx[1] = x[0]+a*x[1]
    dx[2] = b+x[2]*(x[0]-c)
    return dx

plt.figure(figsize = (9, 9))
x_min = 0.1; x_d = 0.003; x_max = 0.34
y_min = 1; y_d = 0.05; y_max = 5
a_n = np.linspace(x_min, x_max, round((x_max-x_min)/x_d)+1)
c_n = np.linspace(y_min, y_max, round((y_max-y_min)/y_d)+1)
map_Rossler = np.zeros((len(c_n), len(a_n)), dtype=int)
for i, c in enumerate(c_n):
    for j, a in enumerate(a_n):
    '''Пропускаем переходной процесс и делаем проверку на разбегание траектории'''
        x0 = 0.001*np.ones(3)
        X = solve_ivp(Rossler, (0, 20000), x0, method='LSODA')
        amp = (max(X.y[0, :])-min(X.y[0, :]))/2
        if abs(amp)>100:
            map_Rossler[i,j] = 0
            continue

        '''Получаем нужный временной ряд'''
        x0 = X.y[:, -1]
        t0 = np.linspace(0, N, round(N/dt)+1)
        X = solve_ivp(Rossler, (0, N), x0, t_eval = t0, method = 'LSODA')

        '''Делаем сечение Пуанкаре плоскостью x = 0'''
        y_sec = []
        y = X.y[1,:]
        x = X.y[0,:]
        for i_x in range(1, len(x)):
            if x[i_x-1]<0 and x[i_x]>=0:
                y_mid = y[i_x-1]-x[i_x-1]*((y[i_x]-y[i_x-1])/(x[i_x]-x[i_x-1]))
                y_sec.append(y_mid)
        i_y = 1
        while i_y < len(y_sec) and abs(y_sec[i_y]-y_sec[0])>delta:
            i_y = i_y + 1
        if i_y > maxregime: i_y = maxregime
        map_Rossler[i,j] = i_y
```

```
plt.imshow(map_Rossler, cmap = 'jet', origin='lower',
extent = (x_min, x_max, y_min, y_max),
aspect = x_d / y_d)
plt.colorbar(boundaries=np.arange(-0.5, maxregime+1.5, 1),
ticks=range(0, maxregime+1),
fraction=0.044)
plt.xlabel('a', fontsize=28)
plt.ylabel('c', fontsize=28)
plt.tight_layout()
plt.show()
```

В итоге получим рис. 12.16. Для построения этого рисунка в один поток может понадобиться 12 часов и более.

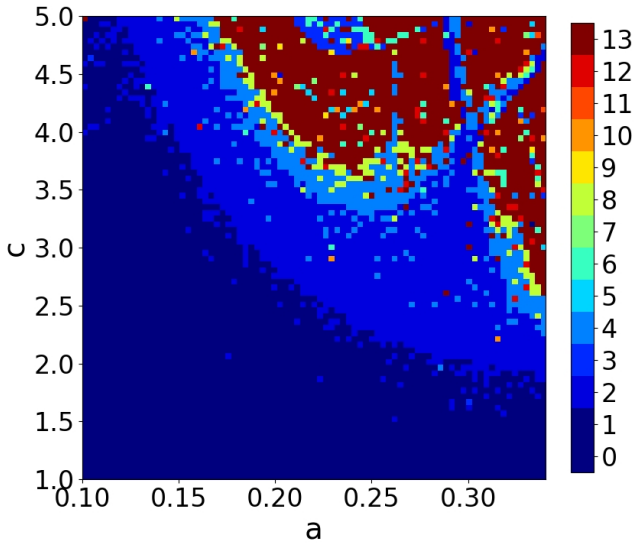


Рис. 12.16. Карта режимов системы Рёсслера на плоскости параметров (a, c). Периодические движения с периодом от 1 до 12 представлены цветами от тёмно-синего до тёмно-красного. Бордовым (цвет 13) обозначены как периодические движения большего периода, так и хаотические режимы.

12.7 Примеры решения заданий

Пример задачи 41 (Символьные вычисления. Модуль sympy.) *Средствами модуля* `sympy` *(функции* `symbols()`, `Eq()` *и* `plot_parametric()`*) построить график фигуры Лиссажу — траектории, прочерчиваемой точкой, совершающей одновременно два гармонических колебания в двух взаимно перпендикулярных направлениях.*

Решение задачи 41 Математическое выражение для фигур Лиссажу выглядит следующим образом:

$$\begin{cases} x(t) = A\sin(at + \delta), \\ y(t) = B\sin(bt), \end{cases}$$

где A, B — амплитуды колебаний, a, b — частоты, δ — сдвиг фаз. Вид кривой сильно зависит от соотношения $a : b$ и значения δ.

Построим (рис. 12.17) фигуру Лиссажу при $A = B = 2$, $\delta = \pi/2$ и соотношении $a : b = 3 : 2$.

```
import numpy as np
from sympy import plot_parametric, symbols, Eq, cos, sin
t=symbols('t')
plot_parametric(2*sin(3*t+np.pi/2),2*sin(2*t),(t,0,2*np.pi),
line_color = 'black', xlabel='x',ylabel='y',
xlim = (-2.5, 2.5), ylim = (-2.5, 2.5))
```

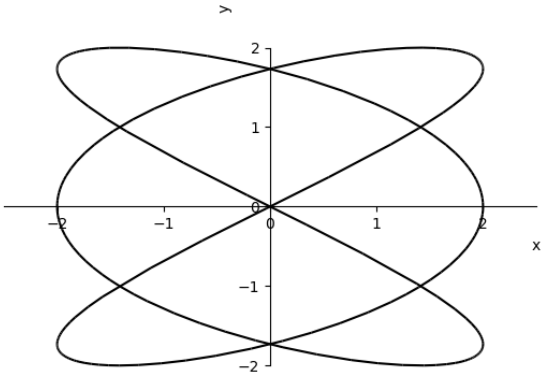


Рис. 12.17. Фигура Лиссажу, построенная средствами модуля `sympy`.

Пример задачи 42 Построим зависимость значения старшего ляпуновского показателя от параметра α для каскадной двумерной системы — отображения Эно (A.36).

Решение задачи 42 Для краткости основные комментарии поместим в код:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']

epslyap = 1e-6
beta = -0.2

def Henon(x, alpha):
    dx = np.zeros(2)
    dx[0] = 1 - alpha*x[0]**2- beta*x[1]
    dx[1] = x[0]
    return dx

alphas = np.arange(0.01, 1.4, 0.01)
N = 500
Lyap = [] #Список значений старшего показателя Ляпунова
for alpha in alphas:
    #Пропускаем переходной процесс
    x = [0.1, 0.1] #начальные условия
    for i in range(10000):
        x = Henon (x, alpha)
    #Вычисляем исходный x и возмущённый x
    X = x
    X_perturbation = np.copy(X)
    X_perturbation[0] += epslyap

    LyapLoc = [] #Список локальных показателей Ляпунова
    for i in range(N-1):
        Xnov = Henon(X, alpha)
        Xnovp = Henon (X_perturbation, alpha)
        delta = np.sqrt(np.sum((Xnovp - Xnov)**2))
        LyapLoc.append(np.log(delta/epslyap))
        X_perturbation = Xnov + (Xnovp - Xnov) * (epslyap/delta)
        X = Xnov
    Lyap.append(np.mean(LyapLoc))

plt.plot(alphas, Lyap, '-', color = 'black')
plt.xlim(0, 1.4)
plt.title('Старший ляпуновский показатель')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\Lambda_1$')
```

```
plt.grid()
plt.show()
```

В итоге получим следующую картинку:

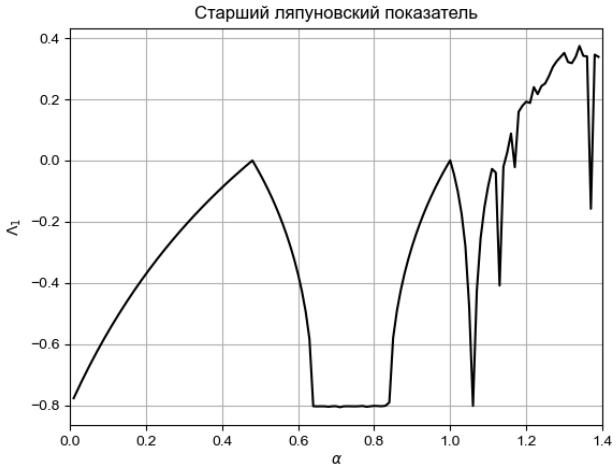


Рис. 12.18. Зависимость значения старшего ляпуновского показателя Λ_1 от параметра α для отображения Эно.

12.8 Задания на исследование динамических систем

Задание 49 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Решите вручную методом Эйлера следующие системы дифференциальных уравнений. Постройте временные реализации и фазовый портрет для вашей системы (если система двумерная, то фазовый портрет рисуйте на осях $2D$; если система трёхмерная, то — на осях $3D$).

1. Осциллятор ван дер Поля (A.8);
2. Осциллятор ван дер Поля - Тоды (A.9);
3. Осциллятор Рэлея (A.10);
4. Генератор с жёстким возбуждением (A.11);
5. Упрощённая модель нейрона ФитцХью-Нагумо (A.12);
6. Полная модель нейрона ФитцХью-Нагумо (A.13);
7. Модель Бонхёффера - ван дер Поля (A.14);

8. Модель Лотки-Вольтерра (A.16);
9. Система Рёсслера (A.17);
10. Система Лоренца (A.18);
11. Генератор Кияшко-Пиковского-Рабиновича (A.19);
12. Генератор Анищенко-Астахова (A.20);
13. Генератор Дмитриева-Кислова с 1.5 степенями свободы (A.21);
14. Система Чуа (A.22);
15. Система ФАПЧ (A.23);
16. Модель Хиндмарш-Роуз (A.24).

Задание 50 *Выполнять одно задание (см. задание 49) с номером* $(n-1)\%m+1$, *где* n *— номер в списке группы, а* m *— число задач в задании.*

Решите вручную методом Рунге-Кутты 4-го порядка. Постройте временные реализации и фазовый портрет для вашей системы (если система двумерная, то фазовый портрет рисуйте на осях $2D$; если система трёхмерная, то — на осях $3D$). Отличаются ли временные реализации, полученные методом Эйлера и методом Рунге-Кутты 4-го порядка?

Задание 51 *Выполнять одно задание (см. задание 49) с номером* $(n-1)\%m+1$, *где* n *— номер в списке группы, а* m *— число задач в задании.*

Решите с помощью встроенных функций `scipy.integrate.odeint` или `scipy.integrate.solve_ivp` следующие системы дифференциальных уравнений. Постройте временные реализации и фазовый портрет для вашей системы (если система двумерная, то фазовый портрет рисуйте на осях $2D$; если система трёхмерная, то — на осях $3D$).

Задание 52 *Выполнять одно задание (см. задание 49) с номером* $(n-1)\%m+1$, *где* n *— номер в списке группы, а* m *— число задач в задании.*

Введите в последнее уравнение вашей системы **запаздывание**. Решите систему методом Эйлера. Постройте временные реализации и фазовый портрет для вашей системы (если система двумерная, то фазовый портрет рисуйте на осях $2D$; если система трёхмерная, то — на осях $3D$).

Задание 53 *Выполнять одно задание (см. задание 49) с номером* $(n-1)\%m+1$, *где* n *— номер в списке группы, а* m *— число задач в задании.*

Добавьте к последнему уравнению вашей системы **белый шум** $\xi(t)$ с параметрами $\mu = 0$ (нулевое среднее), $\sigma = 1$ (среднеквадратичное отклонение). Решите систему методом Эйлера. Постройте временные реализации и фазовый портрет для вашей системы (если система двумерная, то фазовый портрет рисуйте на осях $2D$; если система трёхмерная, то — на осях $3D$).

Задание 54 Постройте семейство резонансных кривых линейного диссипативного осциллятора — зависимость амплитуды вынужденных колебаний от частотной расстройки $A(\delta)$.

Задание 55 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Численно решите (с помощью функции `scipy.optimize.root`, в качестве метода решения укажите метод Крылова) и постройте резонансную кривую осциллятора Дуффинга. Для каждого из трёх начальных условий прорисуйте график своим цветом. Значение параметра интенсивности внешнего воздействия:

1. $P = 2.0$;
2. $P = 2.2$;
3. $P = 2.4$;
4. $P = 2.6$;
5. $P = 2.8$;
6. $P = 3.0$;
7. $P = 3.2$;
8. $P = 3.4$.

Задание 56 Численно (с помощью функции `scipy.optimize.root`) и символьно (средствами `sympy`) решите и постройте семейство резонансных кривых осциллятора Дуффинга для разных значений параметра интенсивности внешнего воздействия в диапазоне $P \in [2.0, 2.8]$ с шагом 0.2. Сравните скорость работы этих двух методов.

Задание 57 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Средствами модуля `sympy` (функции `symbols()`, `Eq()` и либо `plot_implicit()`, либо `plot_parametric()`) построить графики многозначных функций.

1. Окружность $x^2 + y^2 = r^2$, где r — радиус, любое натуральное число;
2. Эллипс $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$, где $a > b$, a, b — любые натуральные числа;
3. Фигура Лиссажу при $A = B = 1$, $\delta = \pi/2$ и соотношении $a : b = 1 : 1$;
4. Фигура Лиссажу при $A = B = 2$, $\delta = \pi/4$ и соотношении $a : b = 1 : 1$;
5. Фигура Лиссажу при $A = B = 1$, $\delta = \pi/2$ и соотношении $a : b = 1 : 2$;
6. Фигура Лиссажу при $A = B = 1$, $\delta = \pi/2$ и соотношении $a : b = 3 : 2$;
7. Фигура Лиссажу при $A = B = 2$, $\delta = \pi/2$ и соотношении $a : b = 3 : 4$;
8. Фигура Лиссажу при $A = 3, B = 1$, $\delta = \pi/2$ и соотношении $a : b = 3 : 4$;
9. Фигура Лиссажу при $A = B = 1$, $\delta = \pi/2$ и соотношении $a : b = 5 : 4$;
10. Фигура Лиссажу при $A = 1, B = 3$, $\delta = \pi/2$ и соотношении $a : b = 5 : 6$.

Задание 58 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Постройте зависимость старшего ляпуновского показателя от параметров системы и бифуркационную диаграмму.

1. кубическое отображение (A.34);
2. отображение окружности (A.35);
3. отображение Эно (A.36);
4. отображение Икеды (A.37);
5. отображение Заславского (A.38).

Глава 13

Параллельное программирование. Модуль `multiprocessing`

13.1 Введение в многопоточные вычисления

13.1.1 О причинах появления многоядерных процессоров

В настоящее время многоядерные процессоры стали доступны практически каждому, в то время как ещё 15 лет назад их можно было встретить только на серверах. Это связано с тем, что повышать производительность оказалось существенно проще за счёт наращивания числа ядер и одновременных потоков вычислений, чем за счёт подъёма тактовой частоты, повышения эффективности процессорного ядра или введения новых инструкций, хотя все перечисленные способы по-прежнему используются. Этим путём пошли оба основных современных производителя микропроцессоров для настольных ПК и ноутбуков: Intel и AMD, а с недавнего прошлого к ним присоединился концерн ARM, разрабатывающий процессоры для смартфонов и планшетных компьютеров (хотя некоторые из них используются и в нетбуках). Поэтому сейчас даже за вполне умеренные деньги нам предлагают приобрести 2, 3, 4, 6 и даже 8 ядерный процессор.

Причины того, почему настольные процессоры стали многоядерными, лежат в возможностях современной индустрии и предпосылки к этому стали формироваться около 25 лет назад. Стоит также сказать, что современные ПК идут всё тем же путём, пусть и с некоторыми вариациями, которым шли суперкомпьютеры уже более полувека. 25–30 лет назад процессоры были ещё достаточно просты и многие не имели даже выделенного блока для операций с вещественными числами, эмулировавшихся с помощью арифметико-логического устройства (АЛУ). Но уже тогда стало ясно, что увеличение самой тактовой частоты — основной способ повышения производительности в первые годы развития `x86`-совместимых процессоров — ограничено возможностями памяти и прочей периферии. Тогда придумали сделать частоты подсистемы памяти и АЛУ различными, но кратными (введение множителя). Это на первых порах дало возможность очень существенно поднять частоты АЛУ, но при этом подсистема памяти стала отставать всё больше и это отставание продолжает накапливаться и сегодня, ведь если память работает на частоте в 10 раз меньше ЦП, данные из неё придётся запрашивать

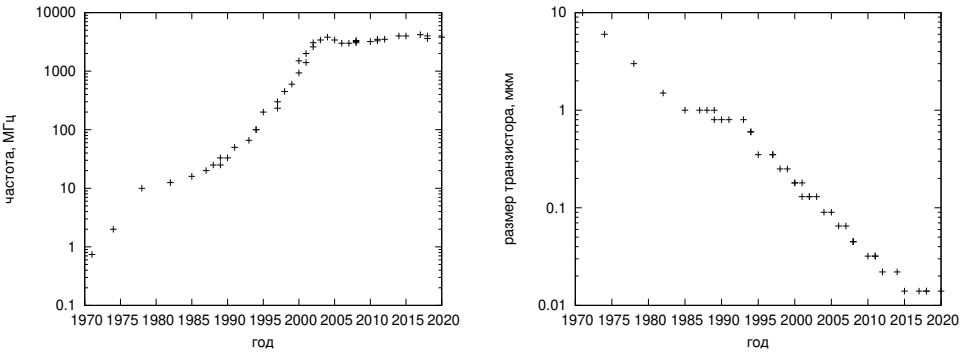


Рис. 13.1. Рост тактовой частоты микропроцессоров корпорации `Intel` со временем (a). Изменение размера транзистора со временем (b).

столь же редко. В результате процессор оказывается на голодном пайке, когда ему просто нечего вычислять, потому что данные из памяти ещё не поступили. Первоначально эта проблема решалась тем, что разрыв был не столь велик, на некоторые операции у ЦП уходило по нескольку тактов, а кроме того были введены дополнительные регистры памяти в самом ЦП, позволявшие хранить дополнительные данные. Но скоро этих полумер стало не хватать.

На рис. 13.1 (a) видно, что в 70-тые рост тактовой частоты микропроцессоров был экспоненциальным (график построен в логарифмическом масштабе), затем замедлился и снова стал экспоненциальным, но уже с меньшим основанием в 90-тые. Однако по достижении максимальной частоты 3.8 МГц в 2004 году рост не только прекратился, но частоты пошли некоторое время вспять, что связано с достижением порога тепловыделения и проблемами утечек. На рис. 13.1 (b) видно, что изменение размера транзистора со временем подчиняется так называемому «закону Мура» и является до сих пор экспоненциальной функцией времени. Освоение новых методов проектирования в конце 80-тых, начале 90-тых привело ко временному нарушению «закона Мура», но было отыграно в дальнейшем. Ступенчатый характер кривой с 1997 года является следствием стратегии `Intel`, которая меняет технический процесс примерно раз в 2 года, таким образом успевая выпустить на одном и том же техническом процессе 2 поколения микропроцессоров.

Понимая, что разрыв по частоте между процессором и памятью становится узким местом — основным источником потери эффективности — производители пошли сразу несколькими путями. Во-первых, ввели дополнительные буферные уровни памяти в самом процессоре, так называемый кэш. Туда попадали данные и инструкции, используемые наиболее часто, чтобы не запрашивать их из памяти каждый раз. Сначала был только кэш 1-го уровня — небольшой объём твердотельной памяти, расположенный в самом процессоре (а в некоторых случаях и на материнской плате) и сделанный вместе с ним из транзисторов того же типа. Потом, по мере того, как память всё более отставала, ввели кэш второго уровня, имеющий бóльшие задержки доступа, но и

больший объём. Сейчас используется уже и кэш третьего уровня, причём суммарный объём кэшей уже иногда превышает объём ОЗУ типичных персональных компьютеров образца 1995–1997 годов. Во-вторых, память заставили работать в 2-х, 3-х и даже 4-х канальном режиме, когда несколько модулей могут быть доступны на запись и чтение одновременно и независимо, что эффективно приводит к увеличению пропускной способности. В-третьих, стали совершенствовать контроллер памяти. Из небольшого простого устройства, располагавшегося на материнской плате, он превратился в чрезвычайно сложного полупроводникового монстра, содержащего миллионы транзисторов. В-четвёртых, перенесли сам контроллер памяти из так называемого «Северного моста» — набора чипов на материнской плате — в сам процессор, что позволило уменьшить задержки доступа.

Само по себе увеличение тактовой частоты не могло длиться бесконечно даже при совершенствовании технических норм производства (см. рис. 13.1 (a)). При достижении величин в 2–3 ГГц начались серьёзные проблемы двух видов: существенно выросло тепловыделение (и энергопотребление, соответственно) и начались квантовые эффекты утечек в транзисторах. Даже регулярное освоение новых технических норм производства, приводящее к уменьшению размера транзисторов (см. рис. 13.1 (b)), не спасало. Поэтому встал вопрос о том, как увеличить КПД на существующих частотах (есть специальный термин — IPS, т. е. instructions per tact — количество инструкций, выполняемых за один период тактового генератора). Для этого, во-первых, придумали и продолжают придумывать новые наборы инструкций, в том числе так называемые «векторные»: `MMX`, `3DNow!`, `SSE`, `SSE2`, `SSE3`, `SSE4`, `AVX` и др. Эти инструкции позволяют сделать за раз несколько действий, например, сложить пару чисел, а потом умножить на третье или сложить три числа и т. п. Но новые инструкции мало придумать и реализовать, нужно ещё заставить разработчиков компиляторов и интерпретаторов языков программирования и, в частности, самой сложной их части — ассемблерных трансляторов — этими инструкциями пользоваться. При этом нельзя забывать и про более старые процессоры, где части этих инструкций может не быть, но на которых новые программы также должны работать. Тогда параллельно был использован второй подход: давайте реализуем одновременно выполнение нескольких стадий одной и той же инструкции, например, запрос адреса и декодирование. Этот подход получил название конвейер. Современные ЦП имеют, как правило, 3-х или 4-путный конвейер, т. е. могут выполнять несколько элементарных операций за такт. Также продублируем некоторые функциональные элементы, например, АЛУ, а данные и инструкции будем загружать из памяти сразу партиями по нескольку штук (суперскалярность). Это уже, фактически, параллелизация, просто сделанная на аппаратном уровне.

Введение конвейера и суперскалярности с одной стороны увеличило производительность в ряде задач, особенно целочисленных, но с другой привело к дополнительным проблемам с памятью, откуда оказалось нужно получать данные для нескольких вычислительных блоков и заранее. Чтобы уметь загружать по несколько инструкций за раз и побыстрее, пришлось ввести дополнительный блок — предсказатель переходов. Этот элемент ЦП пытается с использованием специальных таблиц и вероятностных алгоритмов вычислить, какие инструкции и над какими данными будут выполнены следом за текущими, и на основе такого прогноза контроллер памяти получает задание загрузить соответствующие данные и инструкции в регистры процессора заранее. Несмотря на высокую эффективность предсказателей переходов, дающих верный прогноз в 98% случаев и даже более, оставшиеся случаи оказываются чрезвычайно критичны для про-

изводительности, т. к. приводят к сбросу всего конвейера и необходимости загружать все данные заново. Эти ошибки принято называть «промахами» и они могут привести к простою ЦП в несколько десятков тактов в случае, если необходимые данные имеются в кэшах, и порядка 300 тактов и более, если они доступны только в ОЗУ.

Для более полной загрузки конвейера были придуманы ещё две технологии. Одна из них называется «внеочередное выполнение» и состоит в том, что процессор может переставлять некоторые близкие инструкции местами, выполняя их не в том порядке, в каком они стоят в программе, если это может дать прирост производительности. Например, нам нужно выполнить несколько арифметических операций:

$$
\begin{aligned}
s1 &= a + b \\
s2 &= c + d \\
l1 &= \exp(s1) \\
l2 &= \exp(s2)
\end{aligned}
$$

По правилам процессор должен сначала загрузить значения a и b из памяти, потом, выполнив сложение, аналогично загрузить c и d, наконец, последовательно выполнить два возведения в степень (на самом деле это множество действий). Однако реальный современный процессор, получив результат $s1$, сразу же приступит к третьей операции, благо все имеющиеся данные уже хранятся в регистрах и не зависят от выполнения предшествующей ей второй. Параллельно он запросит c и d из памяти. Таким образом, пока данные будут передаваться из памяти, процессор будет занят полезною работой. Внеочередное выполнение очень помогает во многих современных задачах, написанных на языках высокого уровня прикладными программистами.

Вторая технология имеет различное название у разных производителей и отличается деталями реализации. У `Intel`, которая освоила её первою, она называется `Hyper Threading`. Эта технология позволяет на 1 физическом ядре эмулировать работу 2 логических. При этом польза от такого подхода будет только в том случае, когда, во-первых, один поток вычислений не способен загрузить значительную часть блоков ЦП, а во-вторых, имеется программная поддержка многопоточности. Операционная система видит такой процессор как двухъядерный и не всегда может различить физические и логические ядра.

Все использованные и описанные выше технологии, тем не менее, не позволяют полностью раскрыть возможности современной элементной базы, позволяющей разместить на кристалле всё больше транзисторов. Так, большие кэши способны повлиять на производительность лишь до определённого предела, после которого дальнейшее их наращивание практически бессмысленно или даже вредно. Прочие блоки, как то контроллер памяти, предсказатель переходов, блок внеочередной выборки инструкций и др., нужны в единственном количестве. Увеличивать длину конвейера, дублируя АЛУ и другие основные исполнительные блоки, тоже опасно, поскольку он часто оказывается недозагружен, а промахи становятся более критичны к производительности. Поэтому в определённый момент производители процессоров оказались в ситуации, когда самым удобным и эффективным способом увеличения производительности стало увеличение числа процессорных ядер в рамках одного кристалла.

Однако, чтобы почувствовать прирост производительности, мало просто приобрести многоядерное устройство. Конечно, если у вас одновременно запущено несколько ресурсоёмких задач, например, вы играете в современную компьютерную игру, а в фоне идёт антивирусное сканирование, архивирование большого объёма данных, да

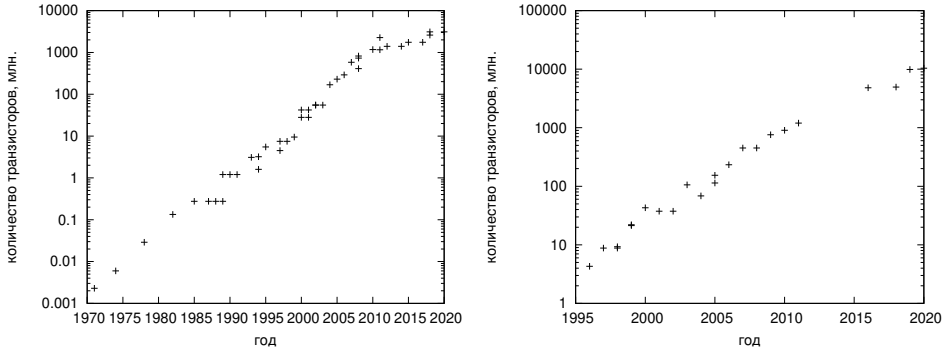


Рис. 13.2. Количество транзисторов в процессорах двух основных производителей: `Intel` — (a) и `AMD` — (b). Видно, что число транзисторов неуклонно растёт по примерно экспоненциальному закону (оба графика построены в логарифмическом масштабе).

ещё и перекодирование недавно снятого на бытовую кинокамеру видео, вы несомненно почувствуете прирост от использования нескольких ядер, поскольку каждая задача сможет использовать как минимум одно отдельное ядро. Но далеко не всегда имеет смысл запускать сразу множество приложений, часто есть одна важная задача, как то же видеокодирование или архивирование, и её желательно максимально ускорить. В такой ситуации всё зависит от того, каким образом написана конкретная используемая вами программа: вы получите прирост от использования многоядерника только если она умеет разделять вычисления на отдельные нити. Когда программа написана за вас, деваться некуда, остаётся только внимательно изучить руководство, чтобы понять, имеет ли смысл бежать в магазин за новейшим многоголовым монстром процессоростроения, или стоит запастись терпением и заняться другими насущными делами, пока ваш компьютер выполняет спланированные вами вычисления (а то и вовсе поставить задачу на выполнение на ночь и ложиться спать). Но если программу пишите вы сами, пусть используя уже готовые библиотеки алгоритмов, можно постараться самому организовать вычисления так, чтобы дополнительные ядра не простаивали. Тому, как организовать это, и посвящена данная книга.

13.1.2 Основные принципы параллелизации

Надо сразу отметить, что далеко не все задачи можно хорошо распараллелить. Классический пример нераспараллеливаемой задачи — вычисление последовательных значений некоторого отображения последования. Возьмём логистическое отображение (A.33):

$$x_{n+1} \quad = \quad rx_n(1-x_n),$$

Пусть стоит задача получить достаточно большое количество последовательных значений при данных r и начальном значении x_0. На первый взгляд кажется, что всё неплохо: действия для получения каждого отдельного значения идентичны, что и нужно для хорошего распараллеливания. Ключевая проблема состоит в том, что каждое последующее значение зависит от предыдущего, а значит и вычислить его нельзя, пока предыдущее не будет получено. Таким образом, мы готовы сформулировать первый принцип параллелизации:

Первый принцип. *Для параллелизации необходимо, чтобы задача была разделима по данным, т. е. чтобы результаты выполнения двух или более операций не зависели друг от друга.*

Представим себе следующую задачу: пусть нам необходимо сложить 4 числа: A, B, C и D. Условимся, что результат должен храниться в S. Обычно эту задачу решают в три этапа:

1. $S = A + B$
2. $S = S + C$
3. $S = S + D$

Количество последовательных действий при этом называют *глубиной* алгоритма. В описанном варианте глубина алгоритма равна 3. Однако такой способ не единственный, можно складывать числа парами, а затем сложить суммы, что наводит на мысль о возможности выполнять сложения пар чисел одновременно, поскольку задача удовлетворяет первому правилу в этой части:

1. $S_1 = A + B$
   $S_2 = C + D$
2. $S = S_1 + S_2$

Количество одновременно выполняемых операций называется *шириной* алгоритма. Для первого способа ширина была равна 1, а для второго стала равна 2, при этом глубина уменьшилась с 3 до 2. Таким образом, задача в принципе параллелизуема частично: 2 из трёх действий мы можем сделать одновременно. Однако на практике, если речь идёт действительно просто о сложении чисел, такой подход малоперспективен. Дело в том, что на выделение некоторых инструкций в отдельную нить, как и на соединение нитей, тратится немало процессорных усилий, гораздо больше, чем необходимо для простого сложения чисел. Из этого вытекает второй принцип параллелизации:

Второй принцип. *В отдельные нити вычислений следует выделять достаточно крупные (затратные) куски кода, иначе польза от задействования дополнительных ядер не будет компенсировать затраты на управление нитями.*

Насколько крупные — однозначно ответить затруднительно. Но, как правило, на создание и объединений нитей современные процессоры тратят единицы и даже десятые доли миллисекунд, поэтому если ваша операция выполняется даже полсекунды и вы можете разделить её пополам, оно того стоит.

Теперь вернёмся к задаче об итерировании (т. е. получении последовательных значений) отображения (A.33). Предположим, что нам нужно насчитать N временных рядов, каждый при своём значении параметра r. Такая задача хорошо параллелизуется, поскольку результаты при каждом r будут полностью независимы от результатов при других. Однако встаёт дополнительный вопрос: сколько потоков вычислений одновременно запускать, если ядер M, а количество возможных параллельных нитей — $N > M$?

Для того, чтобы ответить на этот вопрос, нужно знать, как одноядерные процессоры в прошлом справлялись с несколькими задачами сразу. Все популярные операционные системы современности: `Windows`, начиная с `95`, `Mac OS X`, `Linux`, `BSD` являются многозадачными, т. е. позволяют нескольким приложениям выполняться одновременно. Однако процессор одновременно может выполнять только 1 инструкцию [1]. Поэтому задачи выполнялись на самом деле по очереди, задаче выделялось некоторое время, после чего активная задача становилась пассивною, а активною — какая-то другая. Если время переключения невелико и составляет единицы микросекунд, а большинство задач не дают существенной нагрузки на процессор и просто ожидают действий пользователя, как, например, проводник, у пользователя создаётся полная иллюзия, будто приложения работают одновременно. Но как только какое-либо приложение «отбирает» себе все ресурсы, а остальные остаются на голодном пайке, сразу возникает эффект, известный в народе, как «компьютер тормозит»[2].

На переключение между приложениями, как и на их собственно выполнение, тратятся некоторые ресурсы. Аналогично ресурсы будут расходоваться на переключение между нитями одного приложения. Поэтому наиболее простой ответ на вопрос, сколько нитей должны работать единовременно, сформулирован в третьем принципе:

Третий принцип. *Нужно стараться создать ровно столько вычислительных нитей, сколько активных ядер в системе.*

Следует сразу отметить, что третий принцип не работает, если хотя бы некоторые из созданных нитей не способны загрузить процессор на 100%, например, будучи ограничены скоростью жёсткого диска или канала передачи данных. В такой ситуации число вычислительных нитей стоит ещё увеличить, если это возможно.

13.1.3 Виды многопоточности

Способ, которым приложение способно задействовать несколько ядер, как правило, сильно зависит от специфики данного приложения. Важнейший вопрос состоит в том, могут ли потоки выполнять отдельные действия независимо или должны время от времени обмениваться данными. В первом случае достаточно суметь передать новой нити информацию, необходимую ей для начала вычислений, и забрать результат. Именно такой вид нагрузок часто характерен для научных и инженерных вычислений, задач обработки экспериментальных данных разного рода, особенно если она происходит не в реальном времени. Следует сказать, что такой способ параллельных вычислений также следует считать самым эффективным, поскольку любой обмен данными приводит к потере процессорного времени и снижению производительности.

Для некоторых типов программ, например, для компьютерных игр, обработки данных в реальном времени, работы с базами данных изложенный выше подход неприемлем, для этих типов приложений нити должны уметь обмениваться информацией друг

[1] Строго говоря, в наше время это неверно, часто одновременно выполняются сразу 2–4 действия вследствие наличия конвейера и нескольких АЛУ и блоков работы с плавающей точкой в одном ядре, но было верно, например, для первых Пентиумов.

[2] Аналогичный эффект, к слову, может возникнуть не от перегрузки процессора, а от недостатка ОЗУ, загруженности жёсткого диска, который может не успевать за процессором и памятью, или даже из-за ошибок в программном обеспечении. Поэтому всегда нужно стараться открыть системный монитор (диспетчер задач) и просмотреть, действительно ли виноват процессор.

с другом, что предполагает принципиально иной дизайн приложения, существенно более сложный и гибкий. В данной книге мы будем рассматривать в первую очередь тип нагрузки, характерный для научных и инженерных расчётных задач, когда необходимость коммуникаций между нитями ограничена или вовсе сведена к минимуму.

Вернувшись к последнему примеру с построением набора временных рядов логистического отображения при разных значениях параметра r, можно отметить, что такой способ распараллеливания задачи можно назвать **симметричным**. Приведённый способ разделения задачи часто называют разделением по данным (имеется в виду, что идентичные действия нужно совершать с различными данными). В англоязычной литературе задача разделения по данным часто носит название “scatter–gather problem”. Это значит, что все нити выполняют однотипные операции. Как правило, такой подход — самый эффективный, если речь идёт об обработке больших объёмов данных, поскольку можно добиться близкой к 100%-ной загрузки всех ядер почти на всём времени исполнения.

Но использование симметричной многозадачности (разделения по данным) не всегда возможно или просто организуемо. Тогда прибегают к другому методу, известному как “divide-and-conquer”, суть которого в том, чтобы выделить любые куски кода, которые можно выполнить независимо от предшествующих или последующих. Это могут быть как действия с частично пересекающимися данными, так и полностью независимые операции различного смысла. Далее для простоты будем называть такой подход **несимметричным**.

Рассмотрим следующий пример: нужно оценить дисперсию некоторой случайной величины X по наблюдаемой выборке $\{x_n\}_{n=1}^N$. **Дисперсия** — мера разброса значений случайной величины относительно её математического ожидания. **Математическое ожидание** — среднее (взвешенное по вероятностям возможных значений) значение случайной величины.

Формула для расчёта несмещённой эмпирической дисперсии выглядит следующим образом:

$$\hat{D}_X = \frac{1}{N-1}\sum_{n=1}^{N} x_n^2 - \hat{M}_X^2, \tag{13.1}$$

где $\hat{M}_X$ — эмпирическое среднее (оценка математического ожидания). В свою очередь величина $\hat{M}_X$ находится по формуле:

$$\hat{M}_X = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{13.2}$$

Тогда возникает идея: можно разделить расчёт дисперсии на две независимые операции: в первой оценить величину $\frac{1}{N-1}\sum_{n=1}^{N} x_n^2$, во второй — $\hat{M}_X^2$, а затем просто вычесть из одной другую. Заметим, что обе операции в представленном примере будут совершаться с одними и теми же данными, каждую операцию можно произвести в своей нити. При этом производимые в двух нитях вычисления будут различны по природе и произойдут, скорее всего, за различное время.

При несимметричной многопоточности очень сложно предсказать, какое число нитей будут выполняться одновременно вследствие их различного «веса»: некоторые нити будут достаточно трудоёмки, другие выполнятся сравнительно быстро. Поэтому, если приходится прибегать к такому способу, стараются сделать нитей заметно больше, чем

число ядер в системе, чтобы более равномерно распределить их по ядрам. В такой ситуации одно ядро, например, может успеть обработать только одну нить, другие два — по две, а четвёртое — семь или десять самых малозатратных.

13.1.4 Процессы и потоки, память

Все нити условно принято разделять на 2 типа: процессы и потоки; последние также иногда называют «облегчёнными процессами». Смысл различий главным образом лежит в том, каким образом нитям выделяется память. При старте приложения, будь то браузер, торрент-клиент или среда разработки, автоматически создаётся новый **процесс**. Этот процесс получает определённую область памяти, размеры которой могут меняться со временем, но которая защищена от записи (а часто и для чтения) другими процессами, т. е. переменные, функции и другие объекты, размещённые в этой области памяти, с переменными и функциями других процессов никак не пересекаются. На рис. 13.3 схематично обозначены три процесса, каждый из которых находится в своей области памяти (в своём прямоугольнике). Для обмена данными между процессами существуют специальные механизмы, такие как **сигналы–слоты**, **критические секции**, **разделяемые переменные** и некоторые другие.

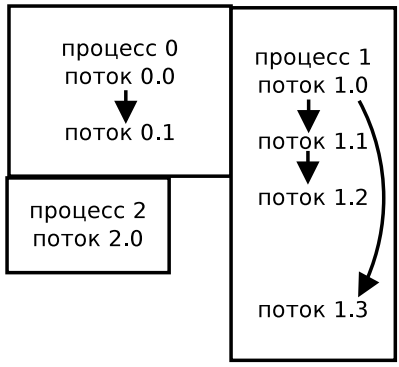


Рис. 13.3. Диаграмма, качественно обозначающая размещение процессов и потоков в памяти компьютера.

При создании процесса создаётся его основной (можно для простоты назвать его нулевым) **поток**. Впоследствии может быть создано ещё несколько потоков внутри того же процесса. Потоки, в отличии от процессов, делят область памяти друг с другом. Поэтому два потока процесса 0 на рис. 13.3 никак не разделены графически. Нулевой поток живёт ровно столько, сколько и сам процесс, но другие потоки могут прекратить своё существование раньше. При создании каждого потока у него есть родитель, например, родителем первого потока является нулевой (потому что больше ничего нет), а родителями последующих может стать любой из имеющихся, зависит это от того, как написана вами программа. Так, например, на представленном рисунке поток 0 процесса 1 стал родителем потоков 1 и 3, а поток 1 — родителем потока 2.

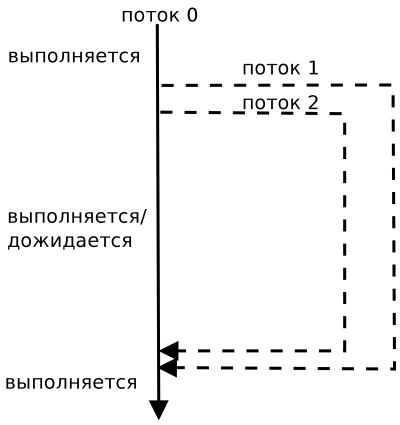


Рис. 13.4. Схема работы трёх потоков: главного и двух дочерних.

Процессы, как и потоки, могут быть порождены из других процессов, являющихся для них родителями. Одно и то же приложение может порождать несколько процессов, например, популярный ныне браузер `Chrome` от `Google` порождает свой процесс на каждую вкладку. В отличие от потоков процессы имеют выделенную область памяти и могут существовать и после смерти родителя. Все процессы можно просмотреть через системный монитор (диспетчер задач)[3] в соответствующем разделе, в то время как потоки увидеть нельзя.

Каждый процесс и каждый поток обязательно имеет свой идентификатор, представляющий собой по сути целое число (хотя по форме он может быть некоторого «странного» типа). Дочерний поток, как правило, имеет механизм памяти о родительском потоке (знает или может узнать его идентификатор), но в родительском потоке программист должен сам позаботиться, чтобы помнить, какие дочерние потоки он породил.

13.1.5 Схема запуска и завершения потоков

При обработке данных, как правило, придерживаются следующей схемы работы с потоками: главный поток запускает на исполнение дочерние, которые производят все необходимые вычисления, после чего ждёт, пока все они корректно не завершатся (см. рис. 13.4). При этом сам главный поток также может выполнят какие-либо полезные действия, если для них ему не требуется знать результаты работы дочерних.

Две наиболее типичные ошибки начинающих при программировании в этом случае заключаются в том, что:

- забывают ждать завершения выполнения дочернего потока в главном, в результате чего главный завершается до того, как дочерние проделают необходимую

[3] В некоторых случаях вам могут потребоваться права администратора, чтобы увидеть некоторые системные процессы.

работу (это может произойти в том числе потому, что забыли сохранить идентификатор дочернего потока);

- запускают и ждут потоки по одному вместо того, чтобы сначала запустить все, а потом дожидаться их завершения, в таком случае программа будет работать корректно и делать, что попросили, но никакого реального распараллеливания не произойдёт, поскольку дочерние потоки одновременно не работают.

Данная схема может повторяться несколько раз в различных вариантах в разных частях программы. Дочерние потоки могут запускаться не все сразу, а по несколько штук (пулами), скажем, по числу активных ядер процессора. Порядок завершения дочерних потоков при этом заранее неизвестен, даже если многопоточность симметричная и потоки делают практически одно и то же. Это обусловлено как наличием в системе дополнительных фоновых процессов, выполняемых на тех же ядрах, так и тем, что операционная система может перераспределять ресурсы по своему усмотрению, так что запущенный вторым поток может завершиться первым и наоборот.

13.2 Параллельное программирование на Python

Для создания дополнительных потоков в рамках текущего процесса в Python используется встроенный (стандартный) модуль `threading`. Однако интерпретатор языка Python устроен таким образом, что параллельное (одновременное) выполнение двух потоков невозможно. Этому есть несколько причин. Первая причина состоит в необходимости отслеживать ссылки для корректной сборки мусора (удаления из памяти неиспользуемых объектов): если с объектами из общей памяти одновременно будут совершать действия различные потоки, никак нельзя гарантировать, что на якобы неиспользуемый объект не появилась новая ссылка. В результате удаления такого объекта могут появиться битые ссылки (указатели в никуда). В языках, где удаление более неиспользуемых данных — работа не специальной подпрограммы–мусорщика, а либо программиста, которому приходится для этого писать отдельный код, как в C и C++, либо компилятора (код для удаления генерируется на этапе компиляции, а не исполнения программы), например, в Rust, а также ограниченно (не для всех типов данных) в Pascal, Fortran, эта проблема отсутствует. Очевидно, что такое решение для интерпретируемых языков нереализуемо. Естественно, за такие плюсы программистам на этих языках приходится платить: программисты на C и C++ часто забывают или неправильно удаляют объекты, что либо приводит к потерям нужных данных во время работы программы, либо к утечкам памяти: выделенная однажды, но уже не используемая, она не возвращается операционной системе и продолжает числиться занятой, пока компьютер не будет перезагружен. В Rust придуман очень сложный и замысловатый синтаксис для описания переменных и передачи их между функциями, столь запутанный, что это стало основным препятствием к распространению этого перспективного языка. В Pascal и Fortran всё хорошо, пока порождённые динамические объекты не содержат кода для порождения других таких объектов (например, методов объекта, внутри которых создаются динамические массивы) и к тому же умирают внутри той же подпрограммы, где были рождены; если же они передаются куда-либо по ссылке, их ручное уничтожение часто оказывается необходимо, как в C или C++.

Вторая причина того, что параллельное выполнение двух потоков невозможно, в том, что интерпретатор Python содержит целый ряд объектов вроде адреса текущей директории или списка загруженных модулей, часто называемых «переменными окру-

жения», самовольное изменение которых одним из потоков приведёт к неработоспособности других и даже к краху интерпретатора. Конечно, обе эти проблемы можно обойти, но для этого необходимо использовать сложные и достаточно дорогие с точки зрения ресурсов техники типа семафоров или синхронизации всех потоков. Причём такие техники не дают 100%-ной уверенности в сохранности данных. Поэтому автор языка Гвидо ван Россум пошёл иным путём: многопоточные (конкурентные) программы на Python с использованием модуля `threading` писать можно, но в каждый момент времени будет задействован только один поток, а остальные будут простаивать — такой подход получил название «глобальная блокировка интерпретатора» (Global interpreter lock, GIL). Следует признать, что Python — не единственный язык, где одновременное выполнение нескольких потоков в одном процессе запрещено по соображениям безопасности. Те же проблемы имеются, например, в языке Ocaml — достаточно популярном нишевом языке, используемом при разработке компиляторов (например, первый компилятор Rust был написан на Ocaml) и в финансовых приложениях.

Если большинство потоков основное время ничего не делают, а, например, ждут данные по сети или с устройств ввода/вывода, подход с попеременным исполнением потоков отлично работает, позволяя чередовать потоки и эффективно использовать ресурсы процессора: те потоки, у которых есть работа, работают по очереди, большинство же спят в ожидании данных или инструкций. Но если речь идёт о параллельной обработке данных, когда необходимо увеличить производительность именно за счёт одновременного исполнения наиболее трудоёмкой части алгоритма, такой подход не годится, потому что не даёт никаких преимуществ перед последовательным исполнением в один поток, а только отнимает время и силы программиста и компьютера на управление дополнительными потоками. Поэтому, если необходимо реально использовать несколько ядер процессора одновременно для ускорения расчётов, в Python используются отдельные процессы, а вместо модуля `threading` используют модуль `multiprocessing`, содержащий функции и классы с почти идентичным функционалом. Каждый процесс по сути представляет собой отдельный интерпретатор, имеет свою изолированную область памяти, свой сборщик мусора и свои общие переменные.

13.3 Среда программирования и консоль выполнения программы

При выполнении многопроцессорных задач во многих стандартных средах программист может столкнуться с непонятными на первый взгляд проблемами. Одна из причин этого состоит в том, что для удобства отладки и просмотра значений переменных такие среды как `IDLE`, в том числе `IDLEX`, и `Spyder` используют встроенную консоль (терминал) Python, связанную с процессом, запущенным изначально. Это приводит к тому, что в Windows из-за организации процессов в этой системе **дочерние процессы не имеют доступа к встроенному терминалу** и все попытки что-то напечатать из него с помощью стандартной функции `print` ни к чему не приведут. В прямом смысле ни к чему: не только не будет результата — вывода печатаемых значений, но и ошибок тоже не будет. Просто ничего! В `IDLE(X)` эту проблему никак нельзя победить и поэтому использовать эту среду для написания и запуска многопоточных приложений просто не рекомендуется. `Spyder` можно перенастроить: для этого меню `Запуск > Настройки для файла` в разделе `Консоль` выставьте «Выполнить во внешнем системном терминале», а внизу в разделе `Внешний системный терминал` выставьте галочку напротив «Перейти в

консоль Python после выполнения». После этого ваша программа будет выполняться не в красивом встроенном окне справа, а в стандартном чёрном окне, известном ещё со времён MS DOS, зато правильно.

Кроме настройки Spyder можно поставить другую среду, которая сразу выполняет программы во внешнем системном терминале, например, простенький текстовый редактор Geany с сайта https://www.geany.org. Поскольку Geany не поставляется вместе с Python и вообще это универсальный блокнот с подсветкою, могущий использоваться для написания программ на самых разных языках, среду придётся научить видеть интерпретатор. Для этого сначала откройте любой «питоний» файл. Далее идите в меню Сборка > Установить команды сборки и внизу в разделе Выполнить команды в строке под номером 1 замените слово python на полный путь к файлу python.exe, например, в нашем случае это был путь C:\Python3\WPy64-3810\python-3.8.1.amd64\python.exe. При использовании Geany может возникнуть странная ошибка inconsistent use of tabs and spaces (непоследовательное использование табуляций и пробелов), вызванная тем, что часть отступов оформлена с помощью символа табуляции, а другая часть — пробелами. Поскольку по умолчанию Geany использует для отображения табуляции 4 пробела, эта проблема не видна глазом. Чтобы её избежать, зайдите в меню Правка > Настройки > Редактор и во второй вкладке Отступы замените «тип» с «Табуляция» на «Пробелы»; далее перейдите на одну вкладку ниже в Правка > Настройки > Файлы и поставьте галочку напротив «Заменить табуляции пробелами» и жмите «Применить». Теперь меню можно свернуть и следует внести в файл любое изменение (достаточно сделать или удалить пустую строку), тогда файл пересохранится при запуске, все ваши табуляции заменятся пробелами и в будущем эта проблема не возникнет.

В Unix-подобных системах (Linux, BSD, MacOS X) использование IDLE(X) также часто приводит к краху или просто зависанию программы во время выполнения. В отличие от Windows в этих системах программа в Spyder будет работать, но при этом она не будет завершать дочерние процессы и они будут висеть в памяти — менеджер процессов в этих системах устроен иначе, но всё равно неидеальным образом. Поэтому использование Geany в этих системах также рекомендуется. Для настройки просто замените команду python в меню Сборка > Установить команды сборки на python3 (по умолчанию подразумевается вторая версия Python).

В качестве среды исполнения можно также использовать ещё одну программу, поставляемую с Python в сборке WinPython — Pyzo. Эта программа с немного смешным названием (её часто называют «пузом») была специально разработана как первая среда для программирования на Python версии 3. Хотя она не очень популярна и имеет довольно экзотические комбинации клавиш по умолчанию (для запуска всей программы, например, нужно нажать <Ctrl>+<Shift>+E), её встроенный терминал, по крайней мере в Windows 10, нормально поддерживает вывод в многозадачном режиме. В дистрибутивах Linux эта программа часто доступна из стандартного репозитория. Для пользователей Windows она может быть полезна тем, что менеджер пакетов Anaconda, который можно вызывать прямо из встроенного терминала, позволяет поставить дополнительные пакеты даже с правами простого пользователя.

Если простенькие среды вроде Geany вам не подходят, вы можете скачать Community версию самой навороченной среды разработки — PyCharm от компании Jet Brains. Она будет занимать очень много мегабайт на диске (примерно, как весь дистрибутив WinPython) и потреблять нереальное количество оперативной памяти во время работы, но зато обеспечит наилучшую статическую проверку кода, подчёркивая и вы-

деляя почти все возможные ошибки прямо в тексте ещё до запуска программы, что в многопоточных приложениях, где отладка сильно затруднена, может быть нелишне.

13.4 Управление процессами вручную. Класс Process

13.4.1 Создание одного дочернего процесса

В большинстве современных языков программирования встроенные средства позволяют выделить выполнение какой-либо подпрограммы: функции или процедуры в отдельный поток и/или процесс. Это естественный приём проектировщиков языков программирования, направленный на формализацию и минимизацию общения между потоками: чем меньше потоки или процессы пересылают данные и обращаются к данным друг друга, тем ниже вероятность ошибки за счёт одновременного доступа к одним и тем же переменным и накладные расходы на копирования данных из памяти друг друга. Поскольку в Python для параллельного программирования используются именно процессы, необходимо импортировать соответствующий класс из модуля `multiprocessing`:

```
from multiprocessing import Process
```

Названия классов в Python, как правило, пишут с большой буквы, а функций — с маленькой, хотя это договорённость, а не строгое правило.

Чтобы провести первое испытание, нам нужна ещё функция, которая делала бы что-то осязаемое, пускай и бесполезное. Давайте используем просто вызов печати на экран:

```
def f1(x):
    print(x)
```

Передать в функцию аргументы можно при создании процесса: всё равно при создании интерпретатора будет произведено много действий по выделению памяти, созданию переменных окружения. Делается это следующим образом:

```
p = Process(target=f1, args=('Hello, world',))
```

Здесь `p` — это созданный нами процесс (переменная особого типа), а два именованных аргумента позволяют задать имя исполняемой в процессе функции (`target=f1`) и передать ей аргументы — `args=('Hello, world',)`; в приведённом случае аргумент — это традиционная фраза `'Hello, world'`, но может быть что угодно, что по смыслу подходит функции `f1`. Обратите внимание, что аргументы функции *всегда* передаются как кортеж! Даже если у нас один аргумент, надо его завернуть в кортеж длины 1 — для этого и запятая перед закрывающеюся скобкой, иначе с точки зрения Python это не конструктор кортежа, а просто скобки, которые в данном случае ничего не значат.

Когда процесс создан и аргументы функции скопированы, он висит в памяти, но ничего не делает. Чтобы запустить его исполнение, нужно использовать метод `start`: `p.start()`. В нашем случае процесс выполнится очень быстро, но в реальных программах таких процессов будет много, они будут считать что-то осмысленное и их результаты понадобятся для последующих действий главному процессу (родителю). Поэтому важно дождаться завершения процесса: `p.join()`. Метод `join` не позволяет главному процессу выполнять последующие действия, пока дочерний не завершится, послав

соответствующий сигнал. Это гарантирует, что, используя в дальнейшем результаты работы дочернего процесса, главный уверен, что все вычисления закончены, а результаты являются окончательными и не будут меняться дочерним процессом одновременно с использованием их главным.

В Python одна и та же программа может использоваться как главная программа и как модуль. Чтобы понять, является ли запущенная в настоящий момент копия главной программой, можно использовать специальную переменную окружения `__name__` (большинство переменных окружения выглядят немного странно, имея одно или два подчёркивания в начале, а некоторые — и в конце). Это текстовая переменная, принимающая значение `"__main__"`, если это главная программа, и значение, равное имени модуля (имени файла), если этот модуль импортирован из другой программы. Помните, мы уже изучали это в главе Модули? В сложных программах на Python принято (хотя это не закон), все действия за исключением импорта и определения констант выносить либо в функции или классы, либо совершать внутри условного оператора `if __name__ == "__main__":`. Это гарантирует отсутствие ряда ошибок, связанных с неверными переменными окружения. При исполнении многопоточных программ использование такого подхода крайне рекомендуется.

Запишем нашу программу целиком:

```
from multiprocessing import Process
def f1(x):
    print(x)
if __name__ == "__main__":
    p = Process(target=f1, args=('Hello, world',))
    p.start()
    p.join()
```

13.4.2 Создание нескольких процессов вручную

При многопоточном программировании принято все параллельные вычисления производить в дочерних (служебных) потоках, а главный поток во время их работы не используется и просто ждёт их завершения. Поэтому создавать нужно сразу несколько процессов, как это показано в программе ниже:

```
def f2(x, y):
    print(x+y)
if __name__ == "__main__":
    p1 = Process(target=f1, args=(10, 20))
    p2 = Process(target=f1, args=('Александр ', 'Пушкин'))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

Здесь функция `f2` принимает два аргумента и печатает их сумму, причём наша функция **полиморфная** — это значит, что она может применяться к аргументам любых типов, лишь бы для них была определена операция «+». Обратите внимание на то, что оба процесса: и `p1`, и `p2` принимают в качестве «цели» (аргумент `target`) одну и

ту же функцию, а вот аргументы функции (args) — различные. Это нормально, когда используется параллелизм данных.

Будьте внимательны! Обязательно сначала запустить оба процесса, а потом уже ждать их, потому что если запускать и ждать процессы поочерёдно, в каждый момент будет задействовано только одно ядро и никакого параллельного исполнения не получится. При этом и запускать, и ждать можно в произвольном порядке.

13.4.3 Возврат значений из функции. Классы Value, Array, Queue.

Как правило, результаты кода, исполняемого в дочернем процессе, нужны в дальнейшем главному процессу для продолжения вычислений. Если попробовать передать эти результаты из функции с использованием стандартного оператора return, ничего путного не выйдет, поскольку возвращаемое значение будет находиться в памяти того же процесса, что и сама функция, а с окончанием процесса вся эта память будет освобождена и все данные будут уничтожены. Попытки передать результат через объект-аргумент, обычно доступный по ссылке, вроде списка или какого-нибудь класса (в Pascal аналогом является передача из процедуры значения, являющегося её аргументом с ключевым словом var), обречены на провал по той же причине: этот объект только в пределах одного процесса ссылочный, а при порождении дочернего процесса он копируется по содержимому целиком и далее все изменения будут действовать уже в пределах каждого процесса отдельно. Вот фрагмент кода, где дочернему процессу было поручено посчитать сумму элементов списка и добавить результат в конец этого списка, чтобы он стал доступен:

```
from multiprocessing import Process
def fsum(x):
    x.append(sum(x))
if __name__ == "__main__":
    l = list(range(10))
    print(l)
    p1 = Process(target=fsum, args=(l,))
    p1.start()
    p1.join()
    print(l)
    fsum(l)
    print(l)
```

Вывод программы:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 45]
```

Наша функция принимает список, и добавляет к нему сумму элементов. Когда мы запускаем её в отдельном процессе, изменения вносятся не в сам оригинальный список l, а в его копию в дочернем процессе, которая по завершении не сохраняется. Но если запускать в лоб — всё работает. Список печатается трижды: первый раз — после создания, чтобы видеть, что в нём было изначально, второй раз — после завершения дочернего процесса (инструкция p1.join()), чтобы видеть, что после работы дочернего

процесса в нём ничего не изменилось, третий раз — после вызова функции напрямую в главном процессе, чтобы видеть, что функция сама по себе написана верно и добавление суммы элементов в конец списка таки происходит.

Чтобы обойти механизмы изоляции памяти, для обмена данными можно воспользоваться такими «дубовыми» решениями, как запись результата в файл (всё равно текстовый или базу данных) или отправка результата по сети, но это сопряжено со многими дополнительными расходами времени процессора и просто выглядит дико. Вместо этого нужно использовать специальные структуры для обмена данными, предусмотренные в Python — это **разделяемые**, или **синхронные** переменные типов `multiprocessing.Value` и `multiprocessing.Array`. Для примера рассчитаем сумму элементов списка в отдельном потоке:

```
from multiprocessing import Process, Value
def fsum(x, a):
    a.value = sum(x)
if __name__ == "__main__":
    l = list(range(10))
    v = Value('d')
    p1 = Process(target=fsum, args=(l, v))
    p1.start()
    p1.join()
    print(v.value)
```

Здесь значение типа `Value` — это не просто число, а специальный объект-контейнер. У этого объекта есть поле — `value`, хранящее нужное нам число. Тип числа указывается при создании объекта, в нашем случае — `'d'` значит `double`, то есть вещественное число двойной точности (8 байт). Создавать объект `Value` нужно в главном процессе до создания соответствующего дочернего процесса, поскольку объект должен быть в процесс скопирован при его создании. Теоретически, объект может быть разделён между тремя и более процессами, но так обычно не делают, чтобы не запутаться.

В приведённом примере список `l` и объект-число `v` копируются в память дочернего процесса `p1` при его создании как аргументы функции `fsum`. В самой функции эти значения известны под именами `x` и `a`, соответственно. Если бы функция была просто вызвана в том же потоке, не важно дочернем или главном, `l` и `x` представляли бы собой две ссылки (два имени) на один и тот же объект, но при выделении функции в отдельный процесс это не так — это два разных, первоначально (при запуске процесса) идентичных объекта. Функция `fsum` помещает в поле `value` переменной `a` сумму элементов списка `x`, при этом, поскольку объекты `v` и `a` суть синхронные копии, происходит остановка выполнения главного процесса, пока значение не будет синхронизовано (в нашем случае это совершенно неважно с точки зрения производительности, так как главный процесс всё равно ждёт завершения дочернего и ничего не делает). Поэтому при завершении дочернего процесса в `v` оказывается искомое значение. Оно попало туда не в результате возврата из функции или передачи аргумента, а непосредственно в процессе выполнения функции за счёт специального механизма синхронизации объектов, реализованного в классе `Value`.

Аналогичный пример можно привести для класса `Array`:

```
from multiprocessing import Process, Array
```

```
def fsquares(x, a):
    for i, el in enumerate(x):
        a[i] = el**2
if __name__ == "__main__":
    l = list(range(10))
    arr = Array('d', len(l))
    p1 = Process(target=fsquares, args=(l, arr))
    p1.start()
    p1.join()
    print(arr[:])
```

Приведённая программа в отдельном потоке создаёт массив элементов, являющихся квадратами элементов списка `l`. Массиву соответствуют два объекта: `arr` в главном потоке и `a` в дочернем (в функции). При создании объекта — синхронного массива необходимо помнить, что кроме типа данных следует также задать его длину — в приведённом случае для этого используется длина исходного списка. При обращении к элементам массива типа `Array` можно пользоваться квадратными скобками, как и для стандартных списков и кортежей или массивов из `numpy`, но если хочется напечатать все элементы, следует использовать полный срез `[:]`, иначе обращение произойдёт не к элементам, а к классу в целом.

Обращения к разделяемым (синхронным) типам данных `Value` и `Array` кодируются теми же символами, что указаны в таблице «Форматы считывания бинарных файлов» из главы Файлы, кроме `?` и `'s'`. То есть можно использовать `'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'q'`, `'Q'`, `'f'` и `'d'`.

Чтобы не заморачиваться со специальными объектами типа `Value` и `Array` и мочь передавать что угодно из процесса в процесс, можно использовать **очереди** `Queue`. ВАЖНО: теперь у нашей функции ровно 1 аргумент — очередь `q`.

```
from multiprocessing import Process, Queue
def fsquares(q):
    x = q.get()
    y = []
    for el in x:
        y.append(el**2)
    q.put(y)


if __name__ == "__main__":
    l = list(range(10))
    q = Queue()
    q.put(l)
    p1 = Process(target=fsquares, args=(q,))
    p1.start()
    p1.join()
    l2 = q.get()
    print(l2)
```

Итак, в главной программе создаём список `l`, создаём очередь `q`, потом кладём в очередь список с помощью метода `q.put(l)`. В функции список изымаем методом `q.get()`,

потом создаём новый список из квадратов элементов исходного списка и кладём его обратно в очередь q. В главной программе после join вынимаем новый список из очереди и печатаем его на экран.

Вывод программы:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

13.4.4 Создание множества процессов в цикле

Как уже было упомянуто выше, высокая эффективность параллелизации и большая полезность параллельного программирования открываются, когда оказывается возможно одновременно запускать не 2 или 3, а много процессов, каждый из которых выполняет примерно идентичную по сложности работу, чтобы процессы завершились почти одновременно и не пришлось долго ждать один из них. Для иллюстрации этой ситуации хорошо подходит задача о расчёте часто используемой при обработке данных меры — коэффициента корреляции, характеризующего степень линейной зависимости между сигналами при сдвиге. Если сигналы доступны в виде временных рядов $\{x_n\}_{n=0}^{N-1}$ и $\{y_n\}_{n=0}^{N-1}$ — конечных последовательностей из N значений величин x и y, измеренных через равные промежутки времени Δt, то коэффициент корреляции $C_{x,y}(\tau)$ при сдвиге по времени τ (далее будем считать, что τ измеряется в шагах выборки и неотрицательно) определяется по формуле (13.3)

$$C_{x,y}(\tau) = \frac{\sum\limits_{n=\tau}^{N-1}(x_n - \hat{M}_x)(y_{n-\tau} - \hat{M}_y)}{(N-\tau-1)\sqrt{\hat{D}_x\hat{D}_y}}, \tag{13.3}$$

где M_x и M_y суть оценки математического ожидания (эмпирические средние), а D_x и D_y суть оценки дисперсии (эмпирические дисперсии) сигналов x и y соответственно.

На практике интерес часто представляет именно зависимость $C_{xy}(\tau)$, для чего вычисления приходится проводить при значительном количестве разных τ. Обозначим максимальное из них за $\tau_{\max}$, тогда всего нужно произвести $(\tau_{\max}+1)$ вычислений, причём они абсолютно независимы и могут быть сделаны одновременно. Для примера ограничимся подсчётом коэффициента автокорреляции $C_{xx}(\tau)$, когда сигналы x и y совпадают.

Для вычисления значения коэффициента автокорреляции для фиксированного τ напишем следующую функцию:

```
def corr1(q):
    x, tau = q.get()
    N = len(x)
    x1 = x[:N-tau]
    x2 = x[tau:]
    znam = x1.std() * x2.std() * (N - tau - 1)
    q.put(np.sum((x1 - x1.mean()) * (x2 - x2.mean())) / znam)
```

Здесь мы сразу передали параметры через очередь и завели служебные переменные x1 и x2 для исходного ряда, обрезанного с начала и с конца, чтобы воспользоваться

векторными операциями с массивами из numpy и не писать цикл по n. Также мы использовали стандартные методы mean и std для массивов numpy, позволяющие рассчитать величины $\hat{M}_x$ и $\sqrt{\hat{D}_x}$.

Для тестирования результатов попробуем посчитать коэффициент автокорреляции для синусоиды. По определению это должен быть косинус. Чтобы распараллелить программу по τ, проще всего кажется для каждого τ создать отдельный процесс, в котором будет выполнена функция corr1 со своим набором параметров: x передаваться будет всегда одно и то же, а вот tau — разное. Такой подход реализован в листинге ниже:

```
if __name__ == "__main__":
    t = np.linspace(0, 100, 10000, False)
    x = np.sin(2*np.pi*t)
    tmp = time()
    taumax = 100
    taus = list(range(taumax+1))
    procs = []
    ques = []
    for it, tau in enumerate(taus):
        q = Queue()
        q.put((x, tau))
        p = Process(target=corr1, args=(q,))
        ques.append(q)
        procs.append(p)
        p.start()
    cfun = []
    for p, q in zip(procs, ques):
        p.join()
        cfun.append(q.get())
    print(time() - tmp)
    plt.plot(cfun)
    plt.show()
```

Программа имеет два основных цикла for. В первом цикле создаются процессы и связанные с ними очереди. Процессы тут же запускаются на исполнение. Поскольку заранее неизвестно, в каком порядке процессы выполнят работу, удобнее всего использовать тот же порядок ожидания их завершения, что и при создании — это происходит во втором цикле, там же из соответствующих этим процессам очередей извлекаются результаты вычислений и складываются в список cfun. При этом, если часть процессов завершится ранее последующих по списку, это не страшно, так как они просто будут висеть в памяти в ожидании вызова своего метода join, почти не потребляя процессорных ресурсов. Функция time выдаёт текущее время в секундах, разница во времени до запуска и завершения всех дочерних процессов и после используется для определения, как долго (в секундах) работала программа.

Значимый недостаток представленного алгоритма состоит в том, что число создаваемых процессов зависит от taumax и теоретически может быть неограниченно велико. На создание каждого процесса тратятся ресурсы. А работать из них одновременно могут далеко не все, а только число, равное числу потоков вычислений, поддерживаемых операционною системой. Слишком большое число потоков приведёт к исчерпанию опе-

ративной памяти и к загрузке процессоров бессмысленными переключениями между многочисленными процессами, которые формально будут работать одновременно, но фактически процессорное время им будет выделяться поочерёдно малыми порциями, потому что достаточного числа свободных ядер одновременно нет. Это число можно узнать с помощью функции `cpu_count` из модуля `multiprocessing` (см. листинг ниже) и оно либо равно числу процессоров, либо вдвое больше за счёт использования технологий, аналогичных `Hyper Threading`.

```
from multiprocessing import cpu_count
if __name__ == "__main__":
    print(cpu_count())
```

Вывод программы:

```
12
```

В нашем случае для ноутбучного процессора Intel i7 8750H это число равно 12, что немало, но всё же гораздо меньше 100, заданных выше. Зная вывод `cpu_count` (его можно присвоить какой-нибудь переменной), можно породить дочерних потоков ровно столько, сколько поддерживает система, но существующая логика программы будет сломана, поскольку в таком случае число потоков не будет соответствовать числу рассматриваемых сдвигов τ. Таким образом, программа, очевидно, нуждается в переработке.

Очевидно, что если схема «один сегмент данных — один процесс» не работает адекватно, то чтобы исправить ситуацию, нужно заставить каждый процесс обрабатывать сразу несколько сегментов данных. Технически можно предложить два варианта решения проблемы: либо следует нагружать один и тот же процесс несколько раз, давая каждый раз ему на выполнение функцию для одного сегмента данных, как мы делали ранее, либо нужно переписать функцию, выполняемую в процессе так, чтобы она обрабатывала сразу несколько сегментов, тогда запускать её в каждом процессе придётся лишь однажды. Оба эти пути реализуемы в Python, но различными средствами. Если мы хотим создавать процессы вручную, как и ранее, следует переписать исполняемую в процессе функцию:

```
def corrn(q):
    x, taus = q.get()
    N = len(x)
    cfuni = []
    for tau in taus:
        x1 = x[:N-tau]
        x2 = x[tau:]
        znam = x1.std() * x2.std() * (N - tau - 1)
        cfuni.append(np.sum((x1 - x1.mean()) * (x2 - x2.mean())) / znam)
    q.put(cfuni)
```

В новую функцию `corrn` вместо одного значения τ через очередь передаётся набор, который обходится циклом `for`, при этом безразлично, будет ли это список, кортеж или массив, поскольку во всех случаях `for` отработает одинаково. Результаты складываются в список `cfuni`, который опять же через очередь возвращается назад.

Основная программа примет вид:

```
if __name__ == "__main__":
    t = np.linspace(0, 100, 10000, False)
    x = np.sin(2*np.pi*t)
    taumax = 100
    taus = list(range(taumax+1))
    procs = []
    ques = []
    ncpu = cpu_count()
    nruns = ceil(len(taus) / ncpu)
    for i in range(ncpu):
        q = Queue()
        q.put((x, taus[i*ncpu:(i+1)*ncpu]))
        p = Process(target=corrn, args=(q,))
        p.start()
        ques.append(q)
        procs.append(p)
    cfun = []
    for p, q in zip(procs, ques):
        p.join()
        cfun.extend(q.get())
```

Здесь пришлось для удобства ввести две новые переменные: `ncpu` — число поддерживаемых системою потоков вычислений и `nruns` — число сегментов данных, которые будут приходиться на каждый поток. При этом на последний поток может приходиться меньше, поскольку за счёт округления вверх функцией `ceil` из модуля `math` произведение `ncpu*nruns` может быть больше длины списка `taus`, содержащего сдвиги по времени, на которых нужно произвести вычисления.

Если же вы не хотите переписывать функцию `corr1` и приведённое в последнем листинге решение кажется запутанным и несколько искусственным, следует использовать класс `Pool` из всё того же модуля `multiprocessing`, решающий многие проблемы за вас.

13.5 Автоматическое управление процессами. Класс `Pool`

Класс `Pool` берёт на себя бóльшую часть работы, которую мы делали вручную ранее. В то же время, он лишает нас определённой гибкости. С использованием `Pool` можно писать программы только для параллелизации по данным, причём писать их надо в функциональном виде. Чтобы понять, как он работает, давайте сначала напишем непараллельную программу для расчёта корреляционной функции, в которой вычисления при разных сдвигах τ организуем не с помощью цикла `for`, а помощью функции `map`:

```
import numpy as np
from itertools import repeat

def corr1(tup):
    x, tau = tup
    N = len(x)
    x1 = x[:N-tau]
    x2 = x[tau:]
```

```
    znam = x1.std() * x2.std() * (N - tau - 1)
    return np.sum((x1 - x1.mean()) * (x2 - x2.mean())) / znam

if __name__ == "__main__":
    t = np.linspace(0, 100, 10000, False)
    x = np.sin(2*np.pi*t)
    taus = list(range(100))
    cfun = list(map(corr1, zip(repeat(x), taus)))
```

Здесь в функцию `corr1` кортежем передаются всё те же массив и сдвиг, которые распаковываются внутри, а результат возвращается через **`return`**, как это обычно принято. Функция `repeat` нужна для того, чтобы сформировать кортежи значений для разных τ при одном и том же ряде x — она повторяет положенную в неё переменную столько раз, какова длина второго элемента оператора **`zip`**.

Использование **`map`** вместо циклов — это подход из функционального программирования, который был подробно разобран ранее. Он хорош тем, что выделяет запускаемый для каждой итерации код в отдельную программную единицу (в приведённом примере — функцию `corr1`) и гарантирует независимость вызовов функции внутри **`map`** друг от друга (цикл этого гарантировать не может), то есть обеспечивает неизменность данных в итоговом списке — весь список создаётся в конце и на любом промежуточном шаге нельзя поменять результаты предыдущего. Идеи функционального программирования часто используются в параллельном именно в связи с задачею обеспечения целостности данных (чтобы одни потоки или процессы не повредили результаты выполнения других).

Чтобы этот же код выполнялся в многопоточном режиме, достаточно сделать небольшие изменения:

1. добавить в начало строчку с импортом класса `Pool`: **`from multiprocessing import`** `Pool`;
2. где-то до вызова функции **`map`** создать объект типа `Pool`: `p = Pool()`;
3. заменить вызов встроенной функции **`map`** вызовом одноимённого метода класса `Pool`, то есть в нашем случае просто написать `p.`**`map`** вместо **`map`**.

Вот и всё!

13.6 Примеры решения заданий

Пример задачи 43 (Расчёт дисперсии временного ряда) *В начале данной главы была предложена идея произвести расчёт дисперсии сигнала, используя многопоточные вычисления. Используя формулы 13.1 и 13.2, написать программу для расчёта несмещённой эмпирической дисперсии.*

Решение задачи 43 Разделяем расчёт дисперсии на две функции: в первой `square` оцениваем величину $\frac{1}{N-1}\sum_{n=1}^{N} x_n^2$, во второй `expect` — величину $\hat{M}_X^2 = \left(\frac{1}{N}\sum_{n=1}^{N} x_n\right)^2$. Для запуска каждой из функций запускаем свой отдельный процесс. Значения получаем с помощью разделяемых переменных типа `Value`. Когда результаты работы обеих функций переданы в главную программу, то остаётся просто вычесть одно из другого.

```
from multiprocessing import Process, Value
import numpy as np

N = 1000 # Количество точек в ряде
A = 3 # Амплитуда сигнала

def square(x, s):
    s.value = 1/(N-1)*np.sum(x**2)

def expect(x, m):
    m.value = (1/N*np.sum(x))**2

if __name__ == "__main__":
    t = np.linspace(0, 100, N)
    x = A*np.sin(2*np.pi*t)
    S = Value('d')
    M = Value('d')
    p1 = Process(target=square, args=(x,S))
    p2 = Process(target=expect, args=(x,M))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    D = S.value - M.value
    print(D)
```

В нашем случае программа выдаёт:

```
4.499999999999998
```

Для гармонического сигнала дисперсия равна $\frac{A^2}{2}$, где A — амплитуда сигнала. Мы задали амплитуду, равную трём, значит дисперсия у нас должна была получится равной 4.5.

Пример задачи 44 (Расчёт старшего ляпуновского показателя) *Распараллелить расчёт старшего ляпуновского показателя Λ_1 для логистического отображения (A.33) по значению параметра r.*

Решение задачи 44 В разделе 12.4 был представлен пример построения зависимости старшего ляпуновского показателя от значения параметра. Такой расчёт — дело довольно затратное с вычислительной точки зрения, при этом Λ_1 при каждом отдельном значении параметра считаются абсолютно независимо друг от друга. Поэтому задача легко поддаётся распараллеливанию, для чего расчёт для отдельного r выделен в функцию `r1Lyap`, а цикл `for` заменён методом `map` класса `Pool` (в листинге — `p.map`, где `p` — объект класса `Pool`) из модуля `multiprocessing`.

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

```
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
from multiprocessing import Pool
from time import time

epslyap = 1e-6
rlist = np.arange(2.4, 4.0, 0.001)
N = 50
Ntrans = 10000

def r1Lyap(r):
    """ Вычисляет ляпуновский показатель для одного фиксированного значения параметра r "
    # Пропускаем переходной процесс:
    x = 0.1
    for i in range(Ntrans):
        x = r*x*(1-x)
    # Вычисляем исходный x и возмущённый x:
    X = x
    X_perturbation = x + epslyap
    # Считаем локальные показатели Ляпунова и складываем их в список:
    LyapLoc = []
    for i in range(N-1):
        X = r*X*(1-X)
        X_perturbation = r*X_perturbation*(1-X_perturbation)
        delta = abs(X_perturbation - X)
        LyapLoc.append(np.log(delta/epslyap))
        X_perturbation = X + (X_perturbation - X) * (epslyap/delta)
    return np.mean(LyapLoc) # возвращаем среднее значение локальных показателей

if __name__ == "__main__":
    p = Pool()
    temp0 = time()
    Lyap = list(p.map(r1Lyap, rlist))
    temp1 = time()
    print(temp1 - temp0)
    plt.plot(rlist, Lyap, '-', color = 'black')
    plt.xlim(2.4, 4.0)
    plt.title('Старший ляпуновский показатель')
    plt.xlabel('r')
    plt.ylabel(r'$\Lambda_1$')
    plt.grid()
    plt.show()
```

Сравните получившийся рисунок с рис. 12.12. Совпали?

Дополнительно из модуля `time` была импортирована одноимённая функция `time`, выдающая системное время. Разница времени после и до запуска функции `map` даёт нам основное время вычислений. Чтобы понять, какое ускорение дало распараллеливание, можно заменить `p.map` в коде на просто `map`. В нашем случае на шестиядерном

процессоре был достигнут выигрыш в 4.5 раза: 1.49 с во многопоточном режиме против 6.74 с в однопоточном, что следует признать неплохим результатом. А как получилось у вас?

Пример задачи 45 (Построение бифуркационной диаграммы) *Распараллелить расчёт построения бифуркационной диаграммы для логистического отображения (A.33) по значению параметра r.*

Решение задачи 45 Ранее в разделе 12.5 был приведён пример построения бифуркационной диаграммы логистического отображения. Для распараллеливания результата расчёт для отдельного r выделен в функцию `ser1r`, а цикл `for` заменён на функцию `p.map` — метод `map` класса `Pool` из модуля `multiprocessing`.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
from multiprocessing import Pool
from time import time

rList = np.arange(2.4, 4.0, 0.001)
N = 50
Ntrans = 10000

def ser1r(r):
    # Пропускаем переходной процесс:
    x = 0.1
    for i in range(Ntrans):
        x = r * x * (1 - x)
    X = np.zeros(N)
    X[0] = x
    for i in range(N-1):
        X[i+1] = r * X[i] * (1 - X[i])
    return X

if __name__ == "__main__":
    temp0 = time()
    p = Pool()
    Mas = np.array(list(p.map(ser1r, rList)))
    temp1 = time()
    print(temp1 - temp0)

    plt.plot(rList, Mas, ',', color = 'black')
    plt.xlim(2.4, 4.0)
    plt.title('Бифуркационная диаграмма')
    plt.xlabel('r')
    plt.ylabel('x')
    plt.show()
```

Сравните получившийся рисунок с рис. 12.14. Определите какое ускорение дало вам распараллеливание, для этого замените `p.map` в коде на просто `map`. У нас при использовании шестиядерного процессора выигрыш по времени составил 5.33 раза, что очень хорошо.

Пример задачи 46 (Карта режимов отображения Эно) *В главе, посвящённой исследованию динамических систем, была показана программа для построения карты режимов отображения Эно. Карты режимов — почти идеальная задача для распараллеливания. Поэтому попробуем переписать ту программу, используя многопоточные вычисления. И сравним время, затраченное в тот и этот раз на построение карты режимов.*

Решение задачи 46 Чтобы решить задачу, нужно переделать внешний цикл `for` в функцию `map`, а его внутренность — вынести в отдельную функцию.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
from time import time
rcParams['font.sans-serif'] = ['Arial', 'Dejavu Sans']
rcParams['font.size'] = 24
from multiprocessing import Pool

delta = 0.01
maxregim = 13 # максимально разрешимый режим
Ntrans = 10000 #длительность переходного процесса
a_min = 0; a_d = 0.01; a_max = 2
b_min = -0.5; b_d = 0.005; b_max = 0.5
a_n = np.arange(a_min, a_max+delta, a_d)
b_n = np.arange (b_min, b_max+delta, b_d)

def Henon(x, alpha, beta):
    dx = np.zeros(2)
    dx[0] = 1 - alpha*x[0]**2- beta*x[1]
    dx[1] = x[0]
    return dx

def onea(a):
    map_1a = []
    for j, b in enumerate(b_n):
        # Пропускаем переходной процесс
        x = [0.1, 0.1] # начальные условия
        flag = False
        for n in range(Ntrans):
            x = Henon (x, a, b)
            if abs(x[0]) > 10:
                flag = True
                break
```

```
        if not flag:
            y_sec = [x[0]]
            for n in range(maxregim+1):
                x = Henon(x, a, b)
                y_sec.append(x[0])
            i_y = 1
            while i_y < len(y_sec) and abs(y_sec[i_y]-y_sec[0]) > delta:
                i_y = i_y + 1
            if i_y > maxregim: i_y = maxregim
            map_1a.append(i_y)
        else:
            map_1a.append(0)
    return map_1a

if __name__ == "__main__":
    plt.figure(figsize = (9, 9))
    tmp = time()
    p = Pool()
    map_Henon = list(p.map(onea, a_n))
    print(time()-tmp)
    plt.imshow(map_Henon, cmap = 'jet', origin='lower',
           extent = (b_min, b_max, a_min, a_max), aspect = b_d / a_d)
    plt.colorbar(boundaries=np.arange(-0.5, maxregim+1.5, 1),
             ticks=range(0, maxregim+1), fraction=0.044)
    plt.xlabel(r'$\beta$', fontsize=28)
    plt.ylabel(r'$\alpha$', fontsize=28)
    plt.tight_layout()
    plt.show()
```

При указанных выше шагах по параметрам вы должны получить точно такую же картинку, как на рис. 12.15. Какой выигрыш по времени вы получили, распараллелив программу? В нашем случае на шестиядерном процессоре выигрыш по времени по сравнению с последовательным решением, приведённым выше, составил 5 раз, а на 40-ядерном — 24 раза. Эти значения далеки от идеала, но при временах порядка нескольких десятков минут и даже часов может быть очень полезно — всё зависит от шага по параметрам a и b, см. (А.36), и длительности переходного процесса `Ntrans`.

Теперь давайте попробуем, кроме ускорения расчётов, получить лучшее разрешение для карты режимов. Для этого уменьшим шаги по параметрам в 5 раз: `a_d = 0.002; b_d = 0.001`. Программа будет работать существенно дольше, в нашем случае получилось 1 ч 12 мин. В результате получили карту режимов, изображённую на рис. 13.5. Сравните её с рис. 12.15.

13.7 Задания на многопоточные вычисления

Задание 59 *Задание выполняйте полностью.*

Кроме дисперсии в ряде случаев для характеристики распределения случайной величины используют и меры более высоких порядков, например, коэффициент асиммет-

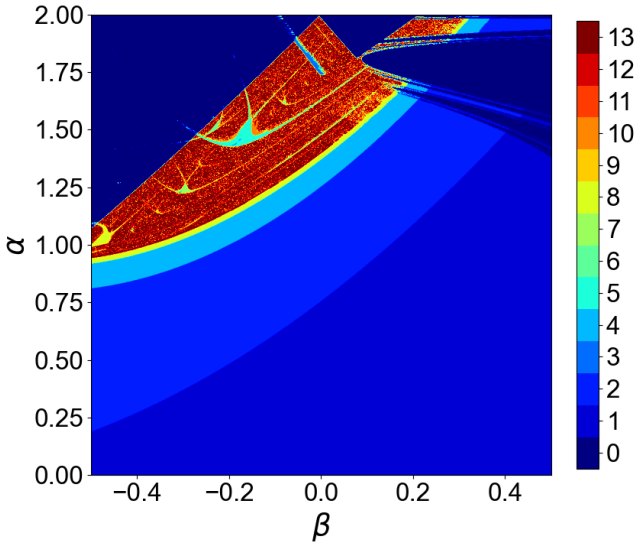


Рис. 13.5. Карта режимов отображения Эно на плоскости параметров (α, β), полученная с хорошим разрешением, благодаря распараллеливанию программы. Периодические движения с периодом от 1 до 12 представлены цветами от тёмно-синего до тёмно-красного. Бордовым (цвет 13) обозначены как периодические движения большего периода, так и хаотические режимы.

рии A_X и эксцесс E_X, определяемые через центральные моменты третьего и четвёртого порядков, соответственно. Формулы для нахождения их эмпирических оценок следующие:

$$\begin{aligned} A_X &= \frac{1}{N-1} \sum_{n=1}^{N} \left(x - M_X\right)^3 / \sigma_X^3 \\ E_X &= \frac{1}{N-1} \sum_{n=1}^{N} \left(x - M_X\right)^4 / \sigma_X^4 - 3 \end{aligned} \tag{13.4}$$

где σ_X — среднеквадратичное отклонение. Попробуйте использовать многопоточность для одновременного вычисления этих характеристик. Выборку генерируйте так же, как и в представленном выше примере для расчёта дисперсии.

Задание 60 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Используя автоматическое управление процессами (класс `Pool`), постройте зависимость ляпуновского показателя от одного из параметров для одного из отображений, описанных в разделе А.2 (значения параметров указаны там же):

1. кубического отображения (А.34) по параметру β;
2. отображения окружности (А.35) по параметру Δ;
3. отображения Эно (А.36) по параметру b;
4. отображения Икеды (А.37) по параметру B;
5. отображения Заславского (А.38) по параметрам Δ, κ при фиксированном $a = 0.3$;
6. отображения Заславского (А.38) по параметрам Δ, a при фиксированном $\kappa = 2$.

Задание 61 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Используя автоматическое управление процессами (класс `Pool`), постройте бифуркационную диаграмму для одного из отображений, описанных в разделе А.2 (значения параметров указаны там же):

1. кубического отображения (А.34) по параметру β;
2. отображения окружности (А.35) по параметру Δ;
3. отображения Эно (А.36) по параметру b;
4. отображения Икеды (А.37) по параметру B;
5. отображения Заславского (А.38) по параметрам Δ, κ при фиксированном $a = 0.3$;
6. отображения Заславского (А.38) по параметрам Δ, a при фиксированном $\kappa = 2$.

Задание 62 Выполнять одно задание с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число задач в задании.

Используя автоматическое управление процессами (класс `Pool`) постройте двумерную карту режимов для одного из отображений, описанных в разделе А.2:

1. кубического отображения (А.34) по параметрам α, β;
2. отображения окружности (А.35) по параметрам Δ, κ;
3. отображения Икеды (А.37) по параметрам A, B при фиксированном $\phi = 0.5$;
4. отображения Заславского (А.38) по параметрам Δ, κ при фиксированном $a = 0.3$;
5. отображения Заславского (А.38) по параметрам Δ, a при фиксированном $\kappa = 2$.

Приложение А

Тестовые динамические системы

Динамическая система представляет собой такую математическую модель некоего объекта, процесса или явления, в которой пренебрегают «флуктуациями и всеми другими статистическими явлениями». Динамическая система описывает (в целом) динамику некоторого процесса, а именно: процесс перехода системы из одного состояния в другое. Фазовое пространство системы – совокупность всех допустимых состояний динамической системы. Таким образом, динамическая система характеризуется своим начальным состоянием и законом, по которому система переходит из начального состояния в другое.

Различают системы с дискретным временем и системы с непрерывным временем. В системах с непрерывным временем, которые традиционно называются **потоками**, состояние системы определено для каждого момента времени на вещественной или комплексной оси. В системах с дискретным временем, которые традиционно называются **каскадами**, поведение системы (или, что то же самое, траектория системы в фазовом пространстве) описывается последовательностью состояний. Каскады и потоки являются основным предметом рассмотрения в символической и топологической динамике.

Динамическая система (как с дискретным, так и с непрерывным временем) часто описывается автономной системой дифференциальных уравнений, заданной в некоторой области и удовлетворяющей там условиям теоремы существования и единственности решения дифференциального уравнения. Положениям равновесия динамической системы соответствуют особые точки дифференциального уравнения, а замкнутые фазовые кривые — его периодическим решениям.

Если на динамику модели влияет любая случайная величина (**шум**), такую модель принято называть **стохастической** вне зависимости от того, каков вклад шумовой добавки. На практике такое определение вызывает большие затруднения при изложении многих вопросов, поскольку, роль шума может быть очень различна и будет зависеть от его интенсивности, коррелированности, закона распределения, способа внесения и очень значительно от свойств той системы, на которую шум влияет. Полезно понимать, что в большинстве случаев имеются две части оператора эволюции: динамическая и стохастическая, относительный вклад которых может быть различен. Для многих автоколебательных систем внесение небольших шумов не приводит к качественному изменению поведения, лишь немного искажая решение. Для ряда систем, например, для линейного осциллятора, внесение шума приводит к изменению типа поведения: появляются новые

колебательные режимы (к этому же классу событий следует отнести явление стохастического резонанса в нелинейных бистабильных системах), но многие характеристики появившегося под влиянием шума режима, например, частота колебаний, определяются не свойствами шума, а свойствами динамической части оператора эволюции — шум выступает главным образом источником энергии. Таким образом, даже при наличии шума изучение стохастических моделей методами теории динамических систем представляется оправданным. Такие модели также будут рассмотрены в данном разделе в отличие от систем (**случайных процессов**), для которых шум является основным в их динамике.

А.1 Потоковые системы (с непрерывным временем)

А.1.1 Логистическое уравнение (Уравнение Ферхюльста)

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right), \tag{A.1}$$

где параметр r характеризует скорость роста популяции (способность к размножению), параметр K — максимальная ёмкость среды, то есть максимально возможное число особей. В качестве K и r можно подставить любые положительные числа.

Эта динамическая система 1-го порядка нелинейная, но допускает точное (аналитическое) решение, называемое логистическою функцией:

$$x(t) = \frac{Kx_0e^{rt}}{K + x_0\left(e^{rt} - 1\right)},$$

где $x_0 = x(t = 0)$, то есть в начальный момент времени. Поэтому вы всегда можете проверить ваше численное решение, используя эту функцию.

А.1.2 Линейный осциллятор с затуханием под воздействием гармонического сигнала

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2x \quad = \quad F\cos(\Omega t + \phi_0), \tag{A.2}$$

при $\omega_0 = 1$, $\gamma = 0.1$ и параметрах внешнего воздействия амплитуда $F = 1$ и частота $\Omega = 1.2$, начальная фаза ϕ_0 — любая.

А.1.3 Нелинейные осцилляторы с затуханием под внешним воздействием

А.1.3.1 С квадратичной нелинейностью

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2x + \alpha x^2 = F\cos(\Omega t + \phi_0), \tag{A.3}$$

А.1.3.2 с кубической нелинейностью (известный как осциллятор Дуффинга)

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x + \beta x^3 = F\cos(\Omega t + \phi_0), \tag{А.4}$$

где γ, ω_0, F, Ω, ϕ_0 имеют тот же смысл, что и для линейного осциллятора, а параметры α и β отвечают за нелинейность, например, за увеличение коэффициента упругости пружины при сжатии. При $\alpha, \beta << 1$ колебания в такой системе будут близки к линейным (например, можно взять $\alpha = \beta = 0.1$). При «больших» α и β, например, около 1, колебания уже будут существенно нелинейными (не похожи на синусоиду). Такие математические модели описывает пассивный нелинейный контур с диодом вместо конденсатора или грузик на нелинейной пружине, для которой не выполняется закон Гука, находящийся под воздействием внешней вынуждающей силы (периодическое воздействие). Интерес представляет нелинейный резонанс, когда $\Omega \approx 2\omega_0$ для системы (А.3) и $\Omega \approx 3\omega_0$ для (А.4).

Ещё один вариант уравнения (А.4) — система с двухъямным потенциалом, записывается следующим образом:

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} - \omega_0^2 x + \beta x^3 = F\cos(\Omega t + \phi_0), \tag{А.5}$$

причём такая система не может быть получена из (А.4) путём замены коэффициентов, поскольку это бы означало, что частота ω_0 есть мнимая величина, т. к. переход от (А.4) к (А.5) подразумевал бы $\omega_0^2 < 0$. В системе (А.5) возможны колебания возле каждого из двух локальных состояний равновесия (слева и справа от нуля, сама неподвижная точка при $x = 0$ в ней неустойчива), а также колебания большой амплитуды с перескоком из одной ямы в другую и обратно.

А.1.3.3 С гармонической нелинейностью

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 \sin(x) = F\cos(\Omega t + \phi_0), \tag{А.6}$$

здесь сам x часто имеет смысл фазы (угла поворота) и является безразмерным.

А.1.3.4 С потенциалом Тоды (экспоненциальная нелинейность)

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2(1 - e^{-x}) = F\cos(\Omega t + \phi_0), \tag{А.7}$$

здесь также x есть безразмерная величина; потенциал такого осциллятора (экспоненту) можно разложить в ряд по степеням x и получить «бесконечную» нелинейность (только коэффициенты перед всё большими степенями x будут всё меньше), при этом константа (единица) взаимно уничтожается.

А.1.4 Автоколебательные системы второго порядка

Автоколебательные системы — такие, колебания в которых поддерживаются с конечною амплитудой без внешнего воздействия. Если изначальная амплитуда мала — колебания «раскачаются», если велика — частично затухнут.

А.1.4.1 Осциллятор ван дер Поля

$$\frac{d^2x}{dt^2} - \left(r - x^2\right)\frac{dx}{dt} + \omega_0^2 x \quad = \quad 0, \tag{А.8}$$

где r имеет смысл отрицательного линейного трения (источник энергии) и в некоторой степени наследует свойства -2γ из уравнений (Б.1–А.7). Эта система второго порядка по праву считается классической моделью теории колебаний. Как правило, именно на её примере обсуждается проблема возбуждения незатухающих колебаний и стабилизации их амплитуды. При малом r, например $r = 0.1$, существуют низкоамплитудные колебания, близкие по форме к гармоническим. При $r \leq 0$ автоколебаний в системе не будет.

А.1.4.2 Обобщённый осциллятор ван дер Поля–Тоды

$$\frac{d^2x}{dt^2} - (r - x^2)\frac{dx}{dt} + \omega^2(1 - e^{-x}) = 0. \tag{А.9}$$

Эта система при малых x и малых r практически ничем не отличается от осциллятора ван дер Поля, а при больших r (например, $r = 3$) имеет существенно другую форму колебаний.

А.1.4.3 Осциллятор Рэлея

Осциллятор Рэлея — вариант осциллятора ван дер Поля, где энергия передаётся в систему не через координату (напряжение в случае радиотехнического генератора), а через скорость (ток):

$$\frac{d^2x}{dt^2} - \left(r - \left(\frac{dx}{dt}\right)^2\right)\frac{dx}{dt} + \omega_0^2 x \quad = \quad 0. \tag{А.10}$$

Система (А.10) может быть получена из (А.8) путём замены переменных.

А.1.4.4 Генератор с жёстким возбуждением

$$\frac{d^2x}{dt^2} - (r + \mu x^2 - x^4)\frac{dx}{dt} + \omega^2 x = 0. \tag{А.11}$$

В этой системе при $r > 0$ и довольно произвольном μ реализуются режимы автоколебаний, качественно сходные с режимами в осцилляторе ван дер Поля. При $r < 0$ и

$\mu > 0$ существует колебательный режим, который недостижим из малых x (надо сразу подать большую амплитуду — это и называется «жёстким возбуждением»), этот режим сосуществует с режимом затухающих колебаний. В зависимости от того, каковы начальные условия, можно попасть в один из них.

А.1.4.5 Упрощённая модель нейрона ФитцХью–Нагумо

$$\begin{aligned} \varepsilon\frac{dx}{dt} &= x - \frac{x^3}{3} - y, \\ \frac{dy}{dt} &= x + a, \end{aligned} \tag{A.12}$$

где $a = 1.2$, $\varepsilon = 0.1$.

А.1.4.6 Полная модель нейрона ФитцХью–Нагумо

$$\begin{aligned} \varepsilon\frac{dx}{dt} &= x - \frac{x^3}{3} - y, \\ \frac{dy}{dt} &= x - by + a, \end{aligned} \tag{A.13}$$

где $a = 0.8$, $b = 0.3$, $\varepsilon = 0.1$.

А.1.4.7 Генератор Бонхёффера – ван дер Поля (модель нейрона ФитцХью–Нагумо)

$$\begin{aligned} \frac{dx}{dt} &= x(a - x)(x - 1) - y + I_a, \\ \frac{dy}{dt} &= bx - \gamma y, \end{aligned} \tag{A.14}$$

где $a = 0.8$, $b = 0.008$, $\gamma = 0.0033$, $I_a = 0.84$. Изменяя внешний ток I_a, можно менять режимы поведения: при меньших I_a амплитуда будет уменьшаться, а их форма будет становиться ближе к синусоиде, при ещё меньших I_a колебания вообще могут затухать. Параметрами b и γ можно регулировать частоту колебаний (желательно менять их вместе пропорционально).

А.1.4.8 Модель нейрона Моррис–Лекара

$$\begin{aligned} C\frac{dV}{dt} &= I_{ext} - g_L(V - V_L) - g_{Ca}m_\infty(V)(V - V_{Ca}) - g_K n(V - V_K), \\ \frac{dn}{dt} &= \frac{n_\infty(V) - n}{\tau_n(V)}, \end{aligned} \tag{A.15}$$

где V — мембранный потенциал в мВ, отсчитываемый от потенциала покоя; n — активационная переменная для калиевого канала.

$I_{ext} = -15$ мкА/см^2 — приложенный ток; $C = 1$ мкФ/см^2 — ёмкость мембраны; параметры настройки для установившегося режима и постоянной времени: $V_1 = -1$ мВ, $V_2 = 15$ мВ, $V_3 = 2$ мВ, $V_4 = 30$ мВ; равновесный потенциал соответствующих ионных каналов: $V_K = -130$ мВ, $V_L = 50$ мВ, $V_{Ca} = 96$ мВ; проводимость утечки: $g_K = 1.7$ мкСм/см^2, $g_L = 0.5$ мкСм/см^2, $g_{Ca} = 1.2$ мкСм/см^2.

Функции $m_\infty(V)$, $n_\infty(V)$ и $\tau_n(V)$ имеют следующий вид:

$$m_\infty(V) = 0.5\left[1 + \mathrm{th}\left(\frac{V - V_1}{V_2}\right)\right],$$
$$n_\infty(V) = 0.5\left[1 + \mathrm{th}\left(\frac{V - V_3}{V_4}\right)\right],$$
$$\tau_n(V) = \left[\phi\,\mathrm{ch}\left(\frac{V - V_3}{2V_4}\right)\right]^{-1},$$

где $\phi = 0.008$ с$^{-1}$ — опорная частота.

А.1.5 Модель Лотки–Вольтерры

Модель Лотки–Вольтерры также известна как система хищник–жертва, поскольку скорость роста (производная) переменной x (жертва или травоядные) пропорциональна самой этой переменной с коэффициентом α (чем больше травоядных, тем быстрее они размножаются) и обратно пропорциональна произведению xy с коэффициентом β, где y — число хищников, а произведение означает вероятность встречи хищника и жертвы. Скорость роста хищников напротив обратно пропорциональна с коэффициентом γ их числу (они конкурируют за пищу) и прямо пропорциональна кормовой базе, т. е. числу жертв с коэффициентом δ, причём также имеет место произведение xy (вероятность встречи).

$$\begin{aligned} \frac{dx}{dt} &= (\alpha - \beta y)x, \\ \frac{dy}{dt} &= (-\gamma + \delta x)y, \end{aligned} \tag{A.16}$$

Все коэффициенты обязательно положительны по их биологическому смыслу. Например, можно использовать значения $\alpha = 0.9$, $\beta = 0.1$, $\gamma = 0.8$, $\delta = 0.2$.

А.1.6 Автоколебательные системы третьего порядка

А.1.6.1 Система Рёсслера

$$\begin{aligned} \frac{dx}{dt} &= -(y + z), \\ \frac{dy}{dt} &= x + ay, \\ \frac{dz}{dt} &= b + z(x - c), \end{aligned} \tag{A.17}$$

где $a = 0.398$, $b = 2$, $c = 4$ для хаотического режима и $a = 0.3$, $b = 0.2$, $c = 1.5$ для периодического.

А.1.6.2 Система Лоренца

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(r - z) - y, \\ \frac{dz}{dt} &= xy - bz, \end{aligned} \tag{A.18}$$

где $\sigma = 10$, $r = 28$, $b = 8/3$ для хаотического режима.

А.1.6.3 Генератор Кияшко–Пиковского–Рабиновича

$$\begin{aligned} \frac{dx}{dt} &= 2hx + y - gz, \\ \frac{dy}{dt} &= -x, \\ \varepsilon\frac{dz}{dt} &= x - f(z), \end{aligned} \tag{A.19}$$

где $h = 0.075$, $g = 0.93$, $\varepsilon = 0.2$, $f(z) = 8.592z - 22z^2 + 14.408z^3$.

А.1.6.4 Генератор с инерционной нелинейностью Анищенко–Астахова

$$\begin{aligned} \frac{dx}{dt} &= mx + y - xz, \\ \frac{dy}{dt} &= -x, \\ \epsilon\frac{dz}{dt} &= -gz + \theta(x)x^2, \end{aligned} \tag{A.20}$$

где $\epsilon = 1.5$, $m = 1.106$, $g = 0.68$, $\theta(x)$ — функция Хевисайда:

$$\theta(x) = \begin{cases} 0 & \text{, при } x < 0, \\ 1 & \text{, при } x >= 0. \end{cases}$$

А.1.6.5 Генератор Дмитриева–Кислова с 1.5 степенями свободы

$$\begin{aligned} \frac{dx}{dt} &= \left(Mze^{-z^2} - x\right)/T, \\ \frac{dy}{dt} &= x - z, \\ \frac{dz}{dt} &= y - z/Q, \end{aligned} \tag{A.21}$$

где $T = 3$, $M = 26$, $Q = 4.8$.

А.1.6.6 Система Чуа

$$\begin{aligned} \frac{dx}{dt} &= \alpha(y - x - f(x)), \\ \frac{dy}{dt} &= x - y + z, \\ \frac{dz}{dt} &= -\beta y, \end{aligned} \tag{A.22}$$

где $f(x) = cx + 0.5(b - c)(|x + d| - |x - d|) + 0.5(a - b)(|x + 1| - |x - 1|)$, $\alpha = 11$, $\beta = 14$, $a = -0.713$, $b = -0.455$, $c = 4.6$, $d = 7.2$.

А.1.6.7 Модель системы фазовой автоподстройки частоты (ФАПЧ)

$$\begin{aligned} \frac{d\varphi}{dt} &= y, \\ \frac{dy}{dt} &= z, \\ \varepsilon_1\varepsilon_2\frac{dz}{dt} &= \gamma - (\varepsilon_1 + \varepsilon_2)z - (1 + \varepsilon_1\cos\varphi)y \end{aligned} \tag{A.23}$$

где φ — текущая разность фаз подстраиваемого и опорного генератора, γ — начальная частотная расстройка, $\varepsilon_1, \varepsilon_2$ — параметры инерционности фильтров. Применительно к динамике нейрона переменную y можно интерпретировать как описывающую изменение мембранного потенциала, параметры ε_1 и ε_2 позволяют задавать необходимый динамический режим, а γ оказывает воздействие, сходное с воздействием внешнего тока в модели Ходжкина-Хаксли.

Для тестирования данную систему можно брать в четырёх характерных режимах: регулярные колебания ($\gamma = 0.28, \varepsilon_1 = 2, \varepsilon_2 = 10$); пачечные разряды с малым числом импульсов в пачке ($\gamma = 0.075, \varepsilon_1 = 4.5, \varepsilon_2 = 10$); пачечные разряды с большим числом импульсов в пачке ($\gamma = 0.19, \varepsilon_1 = 27, \varepsilon_2 = 10$); квазихаотические колебания ($\gamma = 0.25, \varepsilon_1 = 19.5, \varepsilon_2 = 10$).

При построении графиков не забывайте взять фазу по модулю 2π ($\varphi\%(2\pi)$).

А.1.6.8 Модель нейрона Хиндмарш–Роуз

$$\begin{aligned} \frac{dx}{dt} &= y - ax^3 + bx^2 - z + I_a, \\ \frac{dy}{dt} &= c - dx^2 - y, \\ \frac{dz}{dt} &= r\big(s(x - x_R) - z\big), \end{aligned} \tag{A.24}$$

где при $a = 0.55$, $b = 4.8$, $c = 1$, $d = 7$, $r = 10^{-3}$, $s = 6$, $x_R = -1.6$, $I_a = 4$ будут наблюдаться единичные спайки, а при $a = 1$, $b = 3$, $c = 1$, $d = 5$, $r = 10^{-3}$, $s = 4$, $x_R = -1.6$, $I_a = 1.5$ будут наблюдаться бёрсты.

А.1.7 Системы высших порядков

А.1.7.1 Генератор Дмитриева-Кислова с 2.5 степенями свободы

$$\begin{aligned} T\frac{dx}{dt} + x &= F(z), \\ \frac{d^2y}{dt^2} + \alpha_1\frac{dy}{dt} + y &= x, \\ \frac{d^2z}{dt^2} + \alpha_2\frac{dz}{dt} + \omega^2 z &= \alpha_2\frac{dy}{dt}, \\ F(z) &= Mz\exp(-z^2). \end{aligned} \tag{А.25}$$

Параметры для генераторов с 2.5 степенями свободы для хаотического режима выбирались следующие: $T_1 = 0.8$, $\alpha_{11} = 0.06$; $\alpha_{12} = 0.28$, $\omega_1 = 0.39$, $M_1 = 3.3$, $T_2 = 0.6$, $\alpha_{21} = 0.07$, $\alpha_{22} = 0.33$, $\omega_2 = 0.3$, $M_2 = 3.4$.

А.1.7.2 Двухмассовая модель голосовых связок

$$\begin{array}{l} \frac{dx_1}{dt} = y_1 \\ \frac{dy_1}{dt} = \frac{1}{m_1}\left(-r_1y_1 - k_1\left(x_1\right) - k_c\left(x_1 - x_2\right) + F_1\right) \\ \frac{dx_2}{dt} = y_2 \\ \frac{dy_2}{dt} = \frac{1}{m_2}\left(-r_2y_2 - k_2\left(x_2\right) - k_c\left(x_2 - x_1\right) + F_2\right) \\ \frac{dU_g}{dt} = \frac{1}{L_{g1}+L_{g2}}\left(P_s - \left(R_{k1} + R_{k2}\right)U_g\left|U_g\right| - \left(E_{r1} + E_{r2}\right)U_g\right), \end{array} \tag{А.26}$$

где введён ряд обозначений. A_{g1} и A_{g2} — площади отверстия голосовой щели между первою и второю массами соответственно, A_{g01} и A_{g02} — площади в невозмущённом состоянии, когда $x_1 = 0$ и $x_2 = 0$; l_g — поперечная ширина голосовой щели, таким образом $A_{g1} = A_{g01} + 2l_gx_1$ и $A_{g2} = A_{g02} + 2l_gx_2$; d_1 и d_2 — длины масс; A_1 — эффективная входная площадь на входе из голосовой щели в первый резонатор.

Учтены нелинейные свойства пружин: $k_{1,2} = \kappa_{1,2}x_{1,2}\left(1 + \eta_{1,2}x_{1,2}^2\right)$, k_c — коэффициент упругости пружины, связывающей грузы. При этом в случае смыкания, т. е. когда $x_{1,2} < -\frac{A_{g01,2}}{2l_g}$, функция упругости изменяется и принимает вид:

$$\begin{aligned} k_{1,2} &= \kappa_{1,2}x_{1,2}\left(1 + \eta_{1,2}x_{1,2}^2\right) + \\ &+ h_{1,2}x_{1,2}\left(1 + \eta_{h1,2}x_{1,2}^2\right) \end{aligned}$$

где $h_1, 2$ и $\eta_{h1,2}$ — дополнительные линейные и нелинейные коэффициенты жёсткости, отвечающие за силы упругости, возникающие от столкновения масс.

Параметры диссипации имеют вид: $r_{1,2} = \zeta_{1,2}\sqrt{\frac{m_{1,2}}{k_{1,2}}}$, когда нет смыкания и $r_{1,2} = \zeta_{h1,2}\sqrt{\frac{m_{1,2}}{k_{1,2}}}$ при смыкании соответственно первой или второй пары грузов; $\zeta_{1,2}$ и $\zeta_{h1,2}$

— коэффициенты вязкости. Также введены также следующие гидродинамические параметры:

$$\begin{aligned} R_{k1} &= \tfrac{0.19\rho}{A_{g1}^2}, \quad & R_{k2} &= \frac{\rho\left(\frac{1}{2}-\frac{A_{g2}}{A_1}\right)}{A_{g2}^2} \\ L_{g1} &= \tfrac{d_1\rho}{A_{g1}}, \quad & L_{g2} &= \tfrac{d_2\rho}{A_{g2}} \\ E_{r1} &= 12\nu\tfrac{l_g^2 d_1}{A_{g1}^3}, \quad & E_{r2} &= 12\nu\tfrac{l_g^2 d_2}{A_{g2}^3}, \end{aligned}$$

где ρ — плотность, а ν — вязкость воздуха.

Силы $F_{1,2}$, действующие на грузы со стороны воздуха в голосовой щели определяются давлением на первую P_{m1} и вторую P_{m2} массы, выражаемыми по формулам:

$$\begin{aligned} P_{m1} &= P_s - \tfrac{1}{2}\left(1.37\left(\tfrac{U_g}{A_{g1}}\right)^2 - \left(E_{r1}U_g + L_{g1}\dot{U}_g\right)\right) \\ P_{m2} &= P_{m1} - \tfrac{1}{2}\Big(U_g\left(E_{r1}+E_{r2}\right) + \\ &\qquad + \left(L_{g1}+L_{g2}\right)\dot{U}_g - \rho U_g^2\left(\tfrac{1}{A_{g2}^2} - \tfrac{1}{A_{g1}^2}\right)\Big), \end{aligned}$$

причём все давления отсчитываются от атмосферного (т. е. если, скажем, давление на первый груз P_{m1} равно атмосферному, $P_{m1}=0$). Наконец, сами силы $F_{1,2}$ зависят от того, сомкнуты голосовые связки или нет, и выражаются через $P_{m1,2}$ по формуле:

$$\begin{aligned} A_{g1}>0, A_{g2}>0: &\qquad F_1 = P_{m1}d_1 l_g, \quad F_2 = P_{m2}d_2 l_g \\ A_{g1}<0, A_{g2}>0: &\qquad F_1 = P_s d_1 l_g, \quad F_2 = 0 \\ A_{g1}>0, A_{g2}<0: &\qquad F_1 = P_s d_1 l_g, \quad F_2 = P_s d_2 l_g \\ A_{g1}<0, A_{g2}<0: &\qquad F_1 = P_s d_1 l_g, \quad F_2 = 0 \end{aligned}$$

Типичными являются следующие значения параметров: $m_1 = 0.125$, $m_2 = 0.025$, $l_g = 1.4$, $d_1 = 0.25$, $d_2 = 0.05$, $A_{g01} = A_{g02} = 0.056$, $A_1 = 0.168$, $k_c = 25\cdot 10^3$, $k_1 = 80\cdot 10^3$, $k_2 = 8\cdot 10^3$, $\eta_1 = \eta_2 = 100$, $h_{1,2} = 3k_{1,2}$, $\eta_{h1} = \eta_{h2} = 500$, $\zeta_1 = 0.5\cdot 10^4$, $\zeta_2 = 2.2\cdot 10^4$, $\zeta_{h1} = 0.35\cdot 10^5$, $\zeta_{h2} = 10^5$, $\rho = 1.14\cdot 10^{-3}$, $\nu = 18.2466\cdot 10^{-5}$, $P_s = 7.84\cdot 10^3$.

А.1.7.3 Нейрон Ходжкина–Хаксли

$$\begin{aligned} C\frac{dV}{dt} &= I_{ext} - g_K n^4(V-V_K) - g_{Na}m^3 h(V-V_{Na}) - g_L(V-V_L), \qquad \text{(A.27)} \\ \frac{d\mathbf{x}}{dt} &= \alpha_x(V)(1-x) - \beta_x(V)x. \end{aligned}$$

Здесь V обозначает мембранный потенциал в мВ, отсчитываемый от потенциала покоя; $I_{ext} = -30$ мкА/см^2, $C = 1$ мкФ/см^2, $V_K = -12$ мВ, $V_L = 9$ мВ, $V_{Na} = 115$ мВ, $g_K = 36$ мкСм/см^2, $g_L = 0.3$ мкСм/см^2, $g_{Na} = 120$ мкСм/см^2 имеют тот же смысл, что и для модели Моррис–Лекара (А.15); вектор $x = (n, m, h)$, где n и m — активационные переменные для калиевых и натриевых каналов, h — инактивационная переменная натриевых каналов. Функции $\alpha_x(V)$ и $\beta_x(V)$ для каждого типа каналов имеют следующий

вид:

$$\alpha_m(V) = \frac{0.1(V-25)}{1-\exp\left[-(V-25)/10\right]}, \qquad \beta_m(V) = 4\exp(-V/18),$$
$$\alpha_h(V) = 0.07\exp(-V/20), \qquad \beta_h(V) = \frac{1}{1+\exp\left[-(V-30)/10\right]},$$
$$\alpha_n(V) = \frac{0.01(V-10)}{1-\exp\left[-(V-10)/10\right]}, \qquad \beta_n(V) = 0.125\exp(-V/80).$$

А.1.7.4 Модель нефрона (функциональной единицы почки)

$$\begin{aligned}
\frac{dP_t}{dt} &= \frac{1}{C_{tub}}\left(F_{filt}\left(P_t,r\right) - F_{reab} - \frac{P_t - P_d}{R_{Hen}}\right),\\
\frac{dr}{dt} &= v_r,\\
\frac{dv_r}{dt} &= -kv_r - \frac{P_{eq}\left(P_t,r\right) - P_{av}\left(P_t,r\right)}{\omega},\\
\frac{dx_1}{dt} &= \frac{P_t - P_d}{R_{Hen}} - \frac{3x_1}{T},\\
\frac{dx_2}{dt} &= \frac{3\left(x_1 - x_2\right)}{T},\\
\frac{dx_3}{dt} &= \frac{3\left(x_2 - x_3\right)}{T}.
\end{aligned} \tag{A.28}$$

Здесь P_t — давление в ближнем канале, r — его средний эффективный радиус, V_r — скорость его изменения, переменные x_1, x_2, x_3 введены для описания распространения давления по петле Хенле; среднее время распространения — T. P_{av} и P_{eq} — среднее и равновесное давления крови во входном канале, k — коэффициент, описывающий демпфирующие свойства стенок канала (параметр потерь), R_{Hen} — величина гидродинамического сопротивления петли Хенле, R_a и R_e — гидродинамические сопротивления входящего и выходящего каналов.

Нелинейные функции в (А.28) имеют следующий вид:

$$\begin{aligned}
F_{filt}\left(P_t,r\right) &= \frac{P_a - P_g\left(P_t,r\right)}{R_a(r)} - \frac{P_g\left(P_t,r\right) - P_v}{R_e},\\
P_g\left(P_t,r\right) &= P_a + \frac{R_a(r)\left(P_v - P_a\right)}{R_a(r) + R_eH_a + R_e\left(1-H_a\right)\frac{C_a}{C_e(P_t,r)}},\\
P_{av}\left(P_t,r\right) &= \frac{1}{2}\left(P_a - \left(P_a - P_g\left(P_t,r\right)\right)\frac{\beta R_{a0}}{R_a(r)} + P_g\left(P_t,r\right)\right),\\
P_{eq}\left(P_t,r\right) &= 2.4e^{10(r-1.4)} + 1.6(r-1) +\\
&\quad + \Psi\left(x_3\right)\left(\frac{4.7}{e^{13(0.4-r)}+1} + 7.2(r+0.9)\right),\\
\Psi\left(x_3\right) &= \Psi_{\max} - \frac{\Psi_{\max} - \Psi_{\min}}{1 + e^{\alpha\left(\frac{3x_3}{TF_{Hen0}} - S\right)}},\\
R_a(r) &= R_{a0}\left(\beta + (1-\beta)r^{-4}\right).
\end{aligned}$$

Вместе с системой дифференциальных уравнений необходимо решать алгебраическое уравнение:

$$\begin{aligned}&b\left(R_a(r)+R_eH_a\right)C_e^3+\\&+\left(bR_e\left(1-H_a\right)C_a+a\left(R_a(r)+R_eH_a\right)\right)C_e^2+\\&+\left(aR_e\left(1-H_a\right)C_a+\left(P_t-P_v\right)\left(R_a(r)+R_eH_a\right)+\right.\\&\left.+\left(P_v-P_a\right)R_eH_a\right)C_e+\left(P_v-P_a\right)R_e\left(1-H_a\right)C_a=0,\end{aligned}$$

Эта модель содержит 20 параметров, величины которых считаются постоянными. В литературе используются следующие обоснованные из физиологических соображений значения, соответствующие хаотическому режиму: $C_{tub}=3\,\text{нл/кПа}$, $F_{reab}=0.3\,\text{нл/с}$, $P_d=0.6\,\text{кПа}$, $R_{Hen}=5.3\,\text{кПа}\cdot\text{с/нл}$, $P_a=13\,\text{кПа}$, $P_v=1.3\,\text{кПа}$, $a=22\,\text{Па}\cdot\text{л/г}$, $b=0.39\,\text{Па}\cdot\text{л}^2/\text{г}^2$, $R_e=1.9\,\text{кПа}\cdot\text{с/л}$, $H_a=0.5$, $C_a=54\,\text{г/л}$, $k=0.4\,\text{с}^{-1}$, $\omega=20\,\text{кПа}\cdot\text{с}^2$, $T=16\,\text{с}$, $\Psi_{\max}=0.44$, $\Psi_{\min}=0.2$, $\alpha=160$, $\beta=0.67$, $S=0.19$, $R_{a0}=2.3\,\text{кПа}\cdot\text{с/нл}$.

A.1.8 Генераторы с запаздыванием

A.1.8.1 С квадратичной нелинейностью

$$\varepsilon\frac{dx}{dt} = -x(t)+\lambda-x^2(t-\tau_0), \tag{A.29}$$

где $\varepsilon=20$, $\lambda=1.05$, $\tau_0=100$.

Чтобы получить ряд с запаздыванием необходимо сгенерировать не одно начальное условие для переменной x, а $(\frac{\tau_0}{dt}+1)$ начальных условий, где dt — шаг интегрирования.

A.1.8.2 Система Икеды первого порядка

$$\frac{dx}{dt} = -x(t)+\mu\sin\left(x(t-\tau_0)-x_0\right), \tag{A.30}$$

где $\mu=20$, $\tau_0=2$, $x_0=\pi/3$.

A.1.8.3 Система Мэкки–Гласса

$$\frac{dx}{dt} = -0.1x(t)+\frac{0.2x(t-\tau_0)}{1+x^{10}(t-\tau_0)}, \tag{A.31}$$

где $\tau_0=300$.

A.1.8.4 Осциллятор Ланга-Кобаяши (модель лазера третьего порядка)

$$\begin{aligned} \frac{dR}{dt} &= F(t)R(t) + \nu R(t-\tau_0)cos[\phi(t) - \phi(t-\tau_0) + \Omega\tau_0], \\ R(t)\frac{d\phi}{dt} &= \alpha F(t)R(t) - \nu R(t-\tau_0)sin[\phi(t) - \phi(t-\tau_0) + \Omega\tau_0], \\ T\frac{dF}{dt} &= P - F(t) - [1 + 2F(t)]R^2(t), \end{aligned} \tag{A.32}$$

где $T = 1710$, $\alpha = 5$, $\Omega = -0.192$, $P = 0.6$, $\nu = 0.015$, $\tau_0 = 60.5$ для периодического и $\tau_0 = 63.5$ для хаотического режима.

A.2 Каскадные системы (с дискретным временем)

A.2.1 Логистическое отображение

$$x_{n+1} = rx_n(1 - x_n), \tag{A.33}$$

где $r = 2.6$ для периодического режима, $r = 3.7$ для хаотического режима, $r \in [2.4, 4.0]$ для построения бифуркационной диаграммы.

A.2.2 Кубическое отображение

$$x_{n+1} = \alpha - \beta x_n + x_n^3, \tag{A.34}$$

где $\alpha = 0.2$, $\beta = 1.8$ для периодического режима, $\alpha = 0.4$, $\beta = 2.1$ для хаотического режима, для построения бифуркационной диаграммы: $\alpha = 0$, $\beta \in [0.8, 2.5]$.

A.2.3 Отображение окружности

$$x_{n+1} = x_n + \Delta + \kappa \sin x_n, \tag{A.35}$$

где $\Delta = \pi$, $\kappa = 3$ для периодического режима, $\Delta = \pi$, $\kappa = 2$ для хаотического режима, для построения бифуркационной диаграммы: $\Delta \in [0, 2\pi]$, $\kappa = 2$ или $\Delta = 0.001$, $\kappa \in [1, 4]$.

A.2.4 Отображение Эно

$$\begin{aligned} x_{n+1} &= 1 - \alpha x_n^2 - \beta y_n, \\ y_{n+1} &= x_n, \end{aligned} \tag{A.36}$$

где $\alpha = 0.8$, $\beta = -0.2$ для периодического режима, $\alpha = 1.4$, $\beta = -0.2$ для хаотического режима, для построения бифуркационной диаграммы: $\alpha \in [0, 2]$, $\beta = -0.2$.

А.2.5 Отображение Икеды

$$\begin{aligned} x_{n+1} &= A + B[x_n \cos(x_n^2 + y_n^2 + \phi) - y_n \sin(x_n^2 + y_n^2 + \phi)], \\ y_{n+1} &= B[x_n \sin(x_n^2 + y_n^2 + \phi) + y_n \cos(x_n^2 + y_n^2 + \phi)], \end{aligned} \tag{A.37}$$

где $A = 3.8$, $B = 0.06$, $\phi = 0.5$ для периодического режима, $A = 4$, $B = 0.2$, $\phi = 0.5$ для хаотического режима, а для построения бифуркационной диаграммы: $A = 4$, $B \in [0, 0.2]$, $\phi = 0.5$.

А.2.6 Отображение Заславского

$$\begin{aligned} x_{n+1} &= (x_n + \Delta + \kappa \sin x_n + a y_n) \mod 2\pi, \\ y_{n+1} &= a y_n + \kappa \sin x_n, \end{aligned} \tag{A.38}$$

где $\Delta = \pi$, $\kappa = 2.5$, $a = 0.3$ для периодического режима, $\Delta = \pi$, $\kappa = 2$, $a = 0.3$ для хаотического режима, а для построения бифуркационной диаграммы можно использовать $\Delta \in [0, 2\pi]$, $\kappa = 2$, $a = 0.3$.

Приложение Б

Тестовые стохастические системы

Б.1 Потоковые системы (с непрерывным временем)

Б.1.1 Линейный осциллятор с затуханием под воздействием шума

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x \quad = \quad \xi(t), \tag{Б.1}$$

где собственная частота колебаний $\omega_0 = 1$, коэффициент затухания (диссипации) $\gamma = 0.1$, нормальный шум $\xi(t)$ с параметрами $\mu = 0$ (нулевое среднее), $\sigma = 1$ (среднеквадратичное отклонение).

Б.2 Каскадные системы (с дискретным временем)

Б.2.1 Процесс авторегрессии первого порядка

$$x_{n+1} \quad = \quad \alpha x_n + \xi_n, \tag{Б.2}$$

где $\alpha = 0.99$, ξ_n — нормальный шум с нулевым средним.

Б.2.2 Процесс авторегрессии второго порядка

$$x_{n+1} \quad = \quad \alpha_1 x_n + \alpha_2 x_{n-1} + \xi_n, \tag{Б.3}$$

где $\alpha_1 = 1.99$, $\alpha_2 = -0.99$.

Учебное издание

Серия «Библиотека ALT»

СЫСОЕВА Марина Вячеславовна
СЫСОЕВ Илья Вячеславович

**ПРОГРАММИРОВАНИЕ ДЛЯ «НОРМАЛЬНЫХ»
С НУЛЯ НА ЯЗЫКЕ PYTHON**

В двух частях
Часть 2

Учебник

Ответственный редактор: В.Л. Черный
Оформление обложки: А.С. Осмоловская
Вёрстка: Сысоева М.В., Сысоев И.В.

ООО «Базальт СПО»
Адрес для переписки: 127015, Москва, а/я 21
Телефон: (495)123-47-99. E-mail: sales@basealt.ru
http://basealt.ru

Напечатано с готового оригинал-макета

Подписано в печать 21.04.2023. Формат 70x100/16.
Гарнитура Computer Modern. Печать офсетная. Бумага офсетная.
Усл. печ. л. 14,95. Тираж 999 экз. Изд. номер 025.

Издательство ООО “МАКС Пресс” Лицензия ИД N 00510 от 01.12.99 г.
119992, ГСП-2, Москва, Ленинские горы, МГУ им. М.В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42,
корп. 5, эт. 1, пом. I, ком. 6.3-23Н

По вопросам приобретения обращаться: ООО «Базальт СПО»
(495)123-47-99 E-mail: sales@basealt.ru `http://basealt.ru`