Название книги

С.В. Лемешевский
(sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Oct 23, 2018

Основы языка Python 1

1.1. Первые шаги

1.1.1. Программа на Python с переменными

Рассмотрим пример иллюстрирующий математическую модель описывающий траекторию полета мяча в воздухе. Из второго закона Ньютона, предполагая незначительное сопротивление воздуха, мы можем вывести зависимость вертикального положения y мяча от времени t:

$$y = v_0 t - 0.5 g t^2$$

где v_0 — *начальная скорость*, g — ускорение свободного падения, значение которого положим равным 9.81 м/с ^{2}.

Сценарий. Рассмотрим сценарий на Python для вычисления по простой формуле. Предположим, что сценарий у нас сохранен в виде текста в файле ball.py[1]

```
# -*- coding: utf-8 -*-
# Программа для вычисления положения мяча при вертикальном движении

v0 = 5       # Начальная скорость
g = 9.81     # Ускорение свободного падения
t = 0.6      # Время

y = v0*t - 0.5*g*t**2
```

[1] src-python/ball.py

Разбор сценария. Сценарий на языке Python *— это текстовый файл (в нашем случае ball.py[2]), содержащий некоторые инструкции. Мы можем читать сценарий и понимать, что программа способна делать, но сам сценарий не выполняет никаких действий на компьютере, пока интерпретатор Python не прочитает текст сценария и не преобразует его в некоторые действия.

Когда сценарий запущен в интерпретаторе Python, он разбирается и выполняется построчно. Первые две строки

```
# -*- coding: utf-8 -*-
# Программа для вычисления положения мяча при вертикальном движении
```

являются *комментариями*, т.е. как только встречается символ #, интерпретатор Python воспринимает оставшуюся часть строки как комментарий, пропускает ее и переходит к следующей строки.

Замечание

В первой строке указывается кодировка, в которой сохранен файл сценария (в нашем случае *— это UTF8). Если в сценарии не предполагается использование не ASCII символов, то первую строку можно опустить. Однако, мы можем использовать кириллический текст, поэтому в наших сценариях мы всегда будем указывать кодировку.

Следующие 3 строки, интерпретируемые Pyhton:

```
v0 = 5        # Начальная скорость
g = 9.81      # Ускорение свободного падения
t = 0.6       # Время
```

В Python выражения вида `v0 = 5` известны как операторы *присваивания*. Значение правой части, в нашем слу-

[2] `src-python/ball.py`

чае целое число 5, становится *объектом*, а имя переменной слева *— именованной ссылкой на этот объект. Всякий раз, когда мы запишем v0, Python заменит ее целым значением 5. Выражение v1 = v0 создает новое имя для того же целого объекта со значением 5, а не копирует объект.

Таким образом, прочитав эти строки интерпретатор Python знает три переменных v0, g, t и их значения. Эти переменные используются интерпретатором в следующей строке, фактически реализующей некоторую формулу

```
y = v0*t - 0.5*g*t**2
```

В этой строке Python интерпретирует * как оператор умножения, - *— вычитания, ** *— возведения в степень (естественно, + и \ интерпретируются как операторы сложения и деления соответственно). В результате вычисляется значение по формуле в правой части выражения, которое присваивается переменной с именем y. Последняя строка сценария

```
print y
```

выводит на печать значение переменной y. Таким образом при запуске сценария ball.by[3] на экране будет выведено число 1.2342. В тексте сценария имеются также и пустые строки, которые пропускаются интерпретатором Python. Они добавлены для лучшей читабельности кода.

Запуск сценариев Python. Сценарии Python обычно имеют расширение .py, но это не является необходимым. Как мы уже говорили, рассмотренный выше сценарий сохранен в файле ball.py[4]. Запустить его можно следующей командой:

Terminal

```
python ball.py
```

[3] src-python/ball.py
[4] src-python/ball.py

Такая команда явно указывает, что в качестве интерпретатора файла `ball.py` должен использоваться Python. Также могут задаваться аргументы командной строки, который загружается сценарием.

Команда `python ball.py ...` должна запускаться в консольном окне (терминал в Unix, Командная строка (Command Prompt) в MS Windows).

В случае, когда файлу установлено разрешение на выполнение (команда `chmod a+x ball.py`) в ОС Unix (Linux), сценарий можно запускать командой, сценарий можно запускать командой

Terminal

```
./ball.py
```

В этом случае первая строка сценария должна содержать описание интерпретатора:

```
#!/usr/bin/env python
```

В ОС MS Windows можно писать просто

Terminal

```
ball.py
```

вместо `python ball.py`, если расширение `.py` ассоциировано с интерпретатором Python.

1.1.2. Программа на Python, использующая библиотечные функции

Представим, мы стоим на расстоянии, например, 10 м, и наблюдаете за тем как мяч поднимается в воздух после броска. Прямая соединяющая нас и мяч составляет некий угол с горизонтальной прямой, величина которого возрастает и убывает, когда мяч поднимается и опускается соответственно. Давайте рассмотрим положение мяча в некоторый момент времени, когда высота положения мяча равна 10 м.

Вычислим рассматриваемый нами угол. Перед тем как написать программу, мы должны сформулировать некий

алгоритм, т.е. некий способ выполнения необходимых вычислений. В нашем случае, пусть x — *расстояние от нас до точки подброса мяча, а* y — высота, на которой находится мяч. Таким образом, образуется угол θ с поверхностью земли, где $\mathrm{tg}\,\theta = y/x$. Следовательно, $\theta = \mathrm{arctg}\,(y/x)$.

Напишем сценарий выполняющий эти вычисления. Введем переменные `x` и `y` для координат положения x и y, и переменную `angle` для угла θ. Сценарий сохраним в файл ball_angle.py[5]

```
# -*- coding: utf-8 -*-

x = 10   # горизонтальное положение
y = 10   # вертикальное положение

angle = atan(y/x)

print (angle/pi)*180
```

В этом сценарии строка `angle = atan(y/x)` иллюстрирует вызов *функции* `atan`, соответствующей математической функции arctg, с выражением `y/x` в качестве *аргумента* или *входного параметра*. Отметим, что тригонометрические функции (такие как `atan`) возвращают значение угла в радианах, поэтому для преобразования результата в градусы мы использовали выражение `(angle/pi)*180`.

Теперь, запустив на выполнение сценарий, мы получим ошибку

```
NameError: name 'atan' is not defined
```

Очевидно, интерпретатор Python не распознал функцию `atan`, так как эта функция еще не импортирована в программу. Достаточно большой функционал доступен в интерпретаторе по умолчанию, однако намного больший функционал реализован в *библиотеках* Python. Для того, чтобы активировать использование дополнительного функционала, мы должны его явно импортировать. В Python функция `atan`, как и множество других математических функций, собраны в библиотеке `math`. Такая библиотека в терминах Python называется *модулем*. Чтобы

[5] `src-python/ball_angle.py`

мы могли использовать функцию atan в нашем сценарии, мы должны написать

```
from math import atan
```

Добавив данную строку в начало сценария и запустив его, мы приходим к другой проблеме: переменная pi не определена. Переменная pi, соответствующая значению π, также определена в модуле math, и тоже должна быть импортирована

```
from math import atan, pi
```

Очевидно, что достаточно сложно импортировать поименно все необходимые нам функции и переменные из модуля math, поэтому существует быстрый способ импортировать все из модуля math:

```
from math import *
```

Мы будем использовать такое выражение для импорта, чтобы иметь доступ ко всем общим математическим функциям. Таким образом рабочий сценарий ball_angle.py[6] имеет вид:

```
# -*- coding: utf-8 -*-
from math import *

x = 10   # горизонтальное положение
y = 10   # вертикальное положение

angle = atan(y/x)

print (angle/pi)*180
```

На первый взгляд кажется громоздким использование библиотек, так как мы должны знать какой модуль импортировать, чтобы получить желаемый функционал. Возможно, более удобно иметь доступ ко всему необходимому в любое время. Однако это означает, что мы запол-

[6] src-python/ball_angle.py

ним память программы большим количеством информации, которую мы будем редко использовать для вычислений. Поэтому Python имеет большое количество библиотек, реализующих огромные возможности, из которых мы можем импортировать только необходимые в данный момент.

1.1.3. Программа на Python с векторизацией и построением графиков

Вернемся к задаче, описывающей вертикальное положение y мяча после подбрасывания. Предположим, что нас интересуют значения y в каждую миллисекунду первой секунды полета. Это требует повторения вычисления $y = v_0 t - 0.5gt^2$ тысячу раз.

Также построим график зависимости y от t на отрезке $[0, 1]$. Построение такого графика на компьютере подразумевает рисование прямых отрезков между точками кривой, поэтому нам понадобится много точек, чтобы создать визуальный эффект гладкой кривой. Тысячи точек, которые мы вычислим, нам будет достаточно для этого.

Реализацию таких вычислений и построения графика может быть реализовано следующим сценарием (ball_plot.py[7]):

```
# -*- coding: utf-8 -*-

from numpy import linspace
import matplotlib.pyplot as plt

v0 = 5
g = 9.81
t = linspace(0, 1, 1001)

y = v0*t - 0.5*g*t**2

plt.plot(t, y)
plt.xlabel(u't (c)')
plt.ylabel(u'y (м)')
plt.show()

def height(t):
```

[7] src-python/ball_plot.py

```
    h =  v0*t - 0.5*g*t**2
    return h

h = lambda t: v0*t - 0.5*g*t**2
```

Замечание

В нашем сценарии для символьных аргументов мы использовали префикс u (например, `plt.xlabel(u't (c)')`), чтобы указать, что символы содержатся в кодировке UTF8.

Данный сценарий строит график зависимости вертикального положения мяча от времени (см. рис. :numref:'вычисления из сценария ball.py[8] из раздела 1.1.1 мало изменены, но значение y вычисляется для тысячи точек.

Рассмотрим различия рассматриваемого сценария от предыдущих. Первое отличие это строки, которые могли выглядеть следующим образом:

```
from numpy import *
from matplotlib.pyplot import *
```

Мы видим, что numpy является модулем Python. Этот модуль содержит огромный функционал для математических вычислений, а модуль `matplotlib.pyplot` реализует возможности для построения двумерных графиков. Приведенные выше строки представляют быстрый способ загрузки всего функционала, связанного с вычислениями и построением графиков. Однако, фактически мы используем только несколько функций в нашем сценарии: `linspace`, `plot`, `xlabel` и `ylabel`. Многие разработчики считают, что мы должны импортировать только то, что нам нужно, а не весь возможный функционал:

```
from numpy import linspace
from matplotlib.pyplot import plot, xlabel, ylabel
```

[8] src-python/ball.py

Другие предлагают способ импорта, когда используется префикс для функций модуля

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

...

t = np.linspace(0, 1, 1001)

...

plt.plot(x,y)
```

Мы будем использовать все три способа. В нашем сценарии мы использовали два из них.

```
from numpy import linspace
import matplotlib.pyplot as plt
```

Функция `linspace` принимает три аргумента и, в общем случае, вызывается следующим образом:

```
linspace(start, stop, n)
```

Функция `linspace` генерирует `n` равноотстоящих координат, начинающихся со значения `start`, и заканчивающихся `stop`. Выражение `linspace(0, 1, 1001)` создает 1001 координату от 0 до 1 включительно. Математически это означает, что отрезок $[0, 1]$ разбивается на 1000 равных отрезков и значения координат в этом случае вычисляются следующим образом: $t_i = i/1000,\ i = 0, 1, \ldots, 1000$.

Функция `linspace` возвращает объект класса `array`, т.е. некоторый набор чисел (массив). При выполнении арифметических операций с такими объектами, на самом деле эти операции осуществляются для каждого элемента массива. В результате получается аналогичный массив из 1001 элемента, который сохраняется в переменную `y`. Таким образом `y` также является массивом.

Такой подход вычисления нескольких чисел в одну строку называется *векторизацией*. Этот способ очень удобен, так как сокращает не только количество кода, но

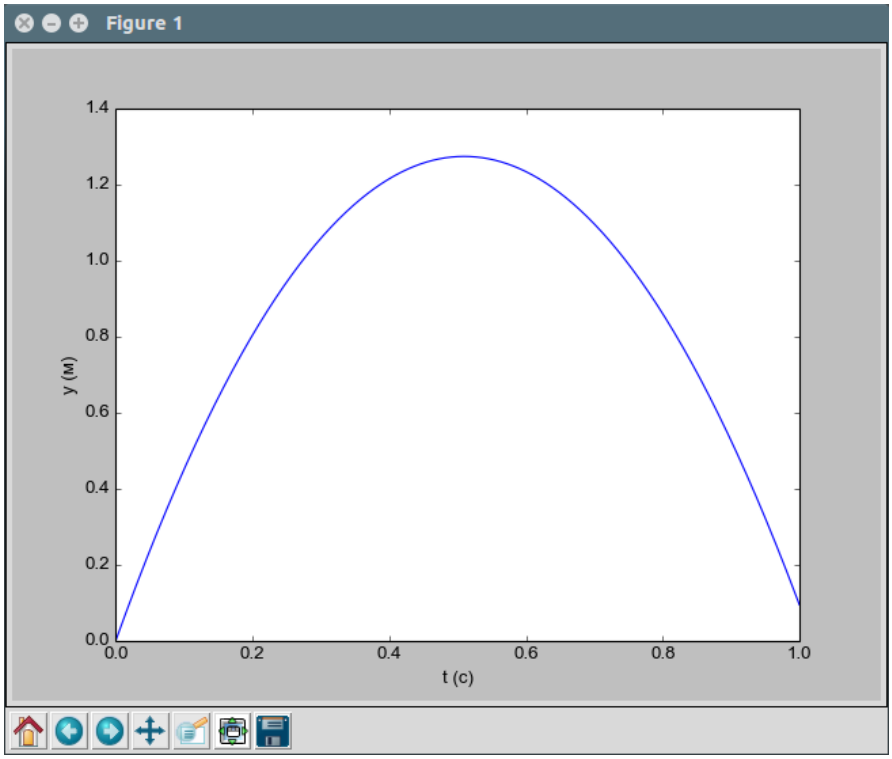

Рис. 1.1 График, построенный с помощью нашего сценария.

и время вычислений по сравнению с использованием циклов `for` или `while`.

Команды для построения графиков достаточно просты:

1. `plot(x, y)` означает построение графика зависимости `y` от `x`
2. `xlabel(u't (c)')` помещает текст `t (c)` на оси `x`.
3. `ylabel(u'y (м)')` помещает текст `y (м)` на оси `y`.

1.1.4. Некоторые основные понятия

Мы рассмотрели несколько простых примеров, иллюстрирующих использование Python для решения простейших математических задач. Прежде чем мы перейдем к более сложным и достаточно реалистичным примерам, рассмотрим некоторые понятия, которые будут часто использоваться в следующих главах: переменные, объекты. Кро-

ме того рассмотрим такие понятия, как ошибки округления, приоритет арифметических операций, деление целых чисел. Также получим чуть больше информации о функционале Python при работе с массивами, построении графиков и выводе результатов.

Интерактивное использование Python. Python можно использовать интерактивно, т.е. мы можем не писать сначала сценарий и запускать его, выполнять операторы и выражения в *оболочке* Python. Мы рекомендуем использовать оболочку IPython (на наш взгляд — лучшая из альтернативных оболочек Python). При использовании IDE Spyder, IPython доступен при запуске в правом нижнем окне. Следуя *подсказке* IPython `In [1]:` (*подсказка* *— это так называемый знак готовности, т.е. программа предлагает вводить команды), мы можем выполнять вычисления

```
In [1]: 2+2
Out[1]: 4

In [2]: 2*4
Out[2]: 8

In [3]: 10/2
Out[3]: 5

In [4]: 4**4
Out[4]: 256
```

Ответу интерпретатора IPython предшествует `Out [q]:`, где `q` *— номер соответствующей входной строки.

Заметим, что, как и в сценарии, можно выполнить команду `from math import *` для использования, например, `pi` или математических функций. Также можно импортировать и другие модули, когда они понадобятся.

Можно также определять переменные и использовать формулы интерактивно:

```
In [1]: v0 = 5

In [2]: g = 9.81

In [3]: t = 0.6

In [4]: y = v0*t - 0.5*g*t**2
```

```
In [5]: print y
1.2342
```

Иногда может понадобиться повторить команду, которую уже выполняли ранее. Для этого можно воспользоваться стрелкой вверх на клавиатуре. Нажав эту клавишу один получим предыдущую команду, два раза *— команду перед этой и т.д. С помощью стрелки вниз мы будем пробегать вперед по истории команд. Также можно поправить выбранную команду перед ее выполнением.

Арифметические операции, круглые скобки и ошибки округления. Как известно арифметические операции `+`, `-`, `*`, `/` и `**` имеют приоритет выполнения. Python обрабатывает операции слева направо, вычисляя один операнд (часть выражения между двумя последовательными `+` или `-`). Внутри каждого операнда операция `**` выполняется перед `*` или `/`. Рассмотрим выражение `x = 2*4**3 + 10*2 - 1.0/4`. В нем три операнда, и Python начинает обрабатывать их слева. В первом операнде `2*4**3` интерпретатор сначала вычислит `4**3`, что даст `64`, затем результат умножается на `2`. Получаем `128`. Следующий операнд `10*2`, т.е. `20`. Затем два операнда складываются, что дает `148`. Результат вычисления последнего операнда равен `0.25`. Таким образом значение переменной `x` становится равным `147.75`.

Отметим, что круглые скобки используются для группировки операндов. Пусть `x = 4` получим результат деления числа `1.0` на `x+1`. Приведем варианты записи в интерпретаторе Python:

```
In [1]: 1.0/x+1
Out[1]: 1.25

In [2]: 1.0/(x+1)
Out[2]: 0.2
```

При первой попытке выполнено сначала деление `1.0` на `x`, к которому затем прибавляется `1`. Python не воспринимает `x+1` как знаменатель. Во второй попытке мы воспользовались круглыми скобками, чтобы сгруппировать зна-

менатель. Так как большинство чисел в памяти компьютера могут быть представлены только приближенно, иногда может получаться не точное значение, что обусловлено *ошибками округления*.

Переменные и объекты. Переменные в Python имеют некоторый *тип*. Если мы запишем `x = 2` в сценарии Python, то `x` становится целой переменной, т.е. переменной типа `int` (целое число). Аналогично, выражение `x = 2.0` означает, что `x` переменная типа `float` (действительное число). В любом случае, Python воспринимает `x` как *объект* типа `int` или `float`. Узнать тип объекта `x` можно с помощью команды `type(x)`. Еще один основной тип переменной *— это тип `str` или строка, используемый для текстовых объектов. Такие переменные задаются двойными или одинарными кавычками:

```
x = ”This is the text”
y = ’This is the text’
```

Оба оператора создают объекты типа `str`.

При необходимости можно *преобразовывать* типы. Если, например, `x` — объект типа `int`, с помощью выражения `y = float(x)` мы создаем объект с плавающей точкой. Аналогично можно получить целое число из типа `float`, воспользовавшись `int(x)`. Преобразование типов может осуществлятся автоматически, как будет показано чуть ниже.

Договоримся, что имена переменных должны быть информативными. При вычислении математических величин, которые имеют стандартные символы, например, α, это следует отразить в имени переменной, используя слово `alpha` в имени переменной в сценарии. Если, например, вычисляем число овец (sheep), то одним из подходящих имен для переменной может быть `no_of_sheep`. Такие имена переменных делают более читабельным код программы. Имена переменных могут содержать цифры и знак подчеркивания, но не могут начинаться с цифры. Буквы могут быть строчными и прописными, в Python они отличаются. Отметим, что некоторые имена в Python зарезервированы и мы не можем их использовать в качестве

имен переменных. Вот некоторые примеры: `for`, `while`, `else`, `global`, `return` и `elif`. Если мы случайно используем зарезервированное слово в качестве имени переменных, то получим сообщение об ошибке.

Выражение `x = 2` присваивает значение `2` переменной `x`. Как можно увеличить `x` на 4? Мы можем воспользоваться выражением `x = x + 4` или выражением (дающим более быстрые вычисления) `x += 4`. Аналогично, `x -= 4` уменьшает значение переменной `x` на 4, `x *= 4` умножает `x` на 4, `x /= 4` делит `x` на 4, обновляя значение переменной `x`.

Что произойдет, если `x = 2`, т.е. объект типа `int`, и мы к нему прибавим 4.0 (объект типа `float`)? Будет иметь место *автоматическое преобразование типов* и новое `x` примет значение `6.0`, т.е. станет объектом типа `float`.

```
In [1]: x = 2

In [2]: type(x)
Out[2]: int

In [3]: x = x +4.0

In [4]: x
Out[4]: 6.0

In [5]: type(x)
Out[5]: float
```

Целочисленное деление. Следует обратить внимание на еще одну проблему *— *целочисленное деление*. Рассмотрим пример деления числа 1 на 4:

```
In [1]: 1/4
Out[1]: 0

In [2]: 1.0/4
Out[2]: 0.25
```

Представлены два способа выполнения этой операции, при этом второй вариант дает правильный результат.

В Python версии 2 первый вариант дает результат так называемого *целочисленного деления*. В этом случае все десятичные знаки отбрасываются, т.е. результат округля-

ется до ближайшего меньшего целого числа. Чтобы избежать этого мы можем явно указать десятичную точку либо в числителе, либо в знаменателе, либо в обоих операндах. Если числитель или знаменатель являются переменными, т.е. мы вычисляем выражение `1/x`, то, чтобы получить результат типа `float`, можно записать `1/float(x)`.

В Python версии 3 операция `/` дает результат типа `float`, а для целочисленного деления используется только операция `//` (`//` есть в Python версии 2).

Форматированный вывод текста и чисел. В результате научных вычислений часто на печать выводится текст, содержащий числа. Естественно желание контролировать формат вывода числа. Например, мы хотим вывести число `1/3` как `0.33` или в виде `3.3333e-1` (3.3333×10^{-1}). Команда `print` — основной инструмент для вывода текста и чисел, обеспечивающий полное управление форматированием. Первый аргумент команды `print` — это строка с определенным синтаксисом, задающим формат вывода, так называемый *printf-синтаксис*. Такое название происходит от функции `printf` из языка программирования C, где такой синтаксис был введен впервые.

Предположим, мы имеем действительное число 12.39486, целое число 23, и текст «сообщение», которые мы хотим вывести на печать двумя способами:

Terminal

```
real=12.394, integer=23, string=сообщение
real=1.239e+01, integer= 23, string=сообщение
```

В первом варианте действительное число выведено в десятичной записи `12.394`, а во втором в научной записи (с плавающей точкой) `1.239e+01`. Целое число в первом варианте выведено в компактном виде, а во втором варианте в текстовом поле из четырех символов.

Следующий сценарий (formatted_print.py[9]) использует `printf`-синтаксис для управления форматированным выводом:

```
# -*- coding: utf-8 -*-
```

[9] src-python/formatted_print.py

```
real = 12.29486
integer = 23
string = 'сообщение'

print "real=%.3f, integer=%d, string=%s" % (real, integer, string)
print "real=%4.3e, integer=%3d, string=%s" % (real, integer, string)
```

Вывод команды `print` — это строка, содержащая текст и набор переменных, вставляемых в текст в местах отмеченных символом `%`. После `%` идет определение формата, например `%f` (для действительных чисел), `%d` (для целых чисел) или `%s` (для строк). Формат вида `%4.3f` означает, что действительное число выводится в десятичной записи, с тремя знаками после запятой, в поле из 4 символов. Вариант `.3f` означает вывод числа в наименьшем возможном поле, в десятичной записи, с тремя знаками после запятой. Замена `f` на `e` или `E` приводит к выводу в научном формате: `1.239e+01` или `1.239E+01`. Запись `%4d` означает вывод целого числа в текстовом поле из четырех символов. Действительные числа также могут выводится форматом `%g`, который используется для автоматического выбора между десятичным и научным форматом, дающим более компактную запись (научный формат удобен для очень маленьких и очень больших чисел).

Типичный пример, когда требуется printf-синтаксис для вывода, когда нужно напечатать выровненные колонки. Предположим мы хотим напечатать колонку значений t и колонку с соответствующими значениями функции $f(t) = t \sin t$. Простейший вариант

```
# -*- coding: utf-8 -*-

from math import sin

t = 0
dt = 0.55

t += dt; f = t*sin(t)
print t, f

t += dt; f = t*sin(t)
print t, f

t += dt; f = t*sin(t)
print t, f
```

дает следующий результат

Terminal

```
Terminal> python2 src-python/unformatted_print.py
0.55 0.287477975912
1.1 0.980328096068
1.65 1.64482729695
```

Обратите внимание на то, что колонки не выровнены, что дает нечитабельный вывод. Добавив в функцию `print` следующий аргумент `'%6.2f %8.3f' %(t, f)`, мы управляем шириной каждой колонки и количеством знаков после запятой:

```
0.55    0.287
1.10    0.980
1.65    1.645
```

Современная альтернатива printf-синтаксису

В современном Python отдается предпочтение использованию метода `format`. Например

```
print 'При t = {t:g} c, y = {y:.2f} м'.format(t=t, y=y)
```

соответствует следующему printf-синтаксису:

```
print 'При t = %g c, y = %.2f м' % (t, y)
```

Область, куда помещается значение переменной, теперь задается фигурными скобками, а в методе `format` перечисляем имена переменных внутри фигурных скобок и соответствующие им имена переменных в программе.

Так как `printf`-синтаксис широко распространен в во многих языках программирования, мы будем использовать его. Однако уже во многих программах на

Python вы можете встретить новый способ форматирования вывода, поэтому мы привели его здесь.

Массивы. В сценарии ball_plot.py[10] из раздела 1.1.3 мы использовали массивы NumPy. Такие массивы создаются и обрабатываются в соответствии с некоторыми правилами, при этом управлять массивами можно как целиком, так и отдельными элементами массива.

Рассмотрим такой пример: предположим у нас есть данные о росте четырех человек. Эти данные мы можем сохранить в массиве с именем `h` следующим образом:

```
h = zeros(4)
h[0] = 1.60
h[1] = 1.75
h[2] = 1.82
h[3] = 1.72
```

Здесь элементами массива являются `h[0]`, `h[1]` и т.д. Самая первая строка в приведенном фрагменте кода

```
h = zeros(4)
```

дает указание Python зарезервировать, или выделить, пространство в памяти для массива `h` из четырех элементов с начальными значениями равными 0. Следующие четыре строки заменяют нулевые значения элементов массива заданными. Элементы, как правило, индексируются с `0`.

Используя двоеточие, мы можем получать *срез* массива. Например, чтобы создать новый массив из двух элементов `h[1]` и `h[2]`, мы можем использовать следующую конструкцию `slice_h = h[1:3]`. Отметим, что определение `1:3` означает индексы `1` и `2`, т.е. последний индекс в определении среза не включается. Полученный массив `slice_h`, индексируется стандартно с 0. Доступ к самому последнему элементу массива можно получить следующим образом `h[-1]`.

[10] `src-python/ball_plot.py`

Копирование массива требует осторожности, так как запись `new_h = h` означает присваивание ссылки, т.е. в этом случае, когда мы будем изменять значения элементов массива `h`, будут изменяться значения соответствующих элементов массива `new_h`.

Построение графиков. Иногда нам нужно построить графики двух функций на одном рисунке. Предположим у нас есть массив `h`, заданный выше и массив `H`. График с двумя кривыми можно построить следующим образом (сценарий plotting_two.py[11]):

```
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt

h = np.zeros(4)

h[0] = 1.60; h[1] = 1.75; h[2] = 1.82; h[3] = 1.72
H = np.zeros(4)
H[0] = 0.60; H[1] = 0.30; H[2] = 1.90; H[3] = 1.99

numbers = np.zeros(4)
numbers[0] = 1; numbers[1] = 2; numbers[2] = 3; numbers[3] = 4

plt.plot(numbers, h, numbers, H)
plt.xlabel(u'Номер по порядку')
plt.ylabel(u'Значение')
plt.show()
```

Результат работы сценария представлен на рисунке

Две кривые на одном графике можно построить и другим способом, используя две команды `plot`. Например, вы можете построить первую кривую

```
plot(numbers, h)
hold('on')
```

Затем можно выполнить различные команды и вычисления в сценарии перед построением второй кривой

```
plot(numbers, H)
hold('off')
```

[11] src-python/plotting_two.py

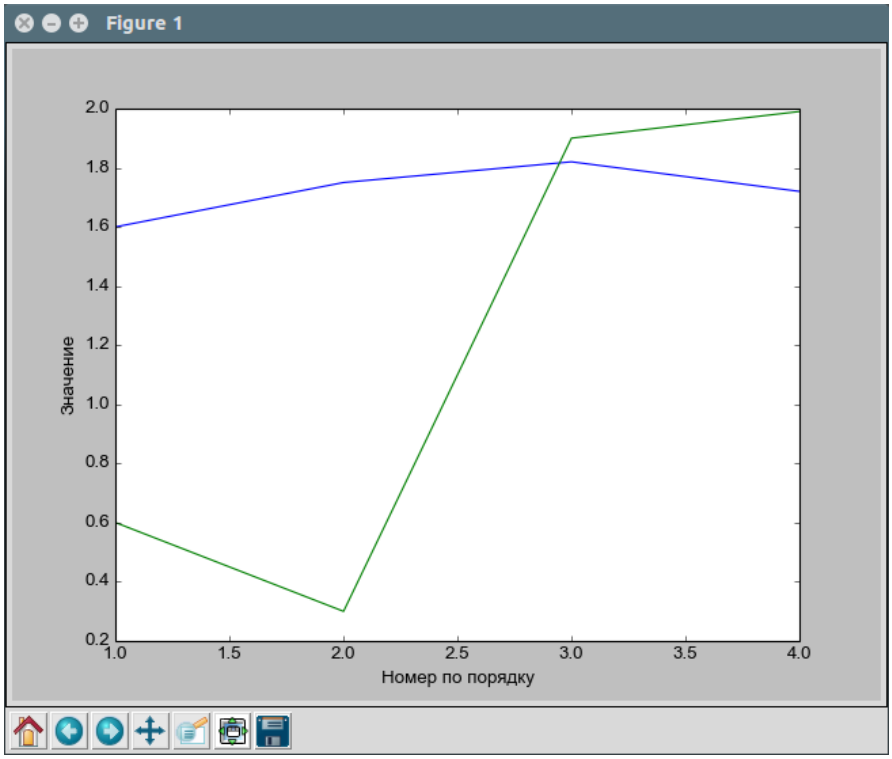

Рис. 1.2 График, построенный с помощью сценария.

Мы здесь использовали функцию `hold`: `hold('on')` сообщает Python, чтобы следующие кривые строились в том же окне. Python выполняет это до тех пор пока не появится команда `hold('off')`. Если не использовать команды `hold('on')` и `hold('off')`, то вторая команда построения кривой перепишет первую, т.е. появится только вторая кривая.

В случае, когда необходимо построить две кривые на отдельных графиках можно построить стандартно первую кривую

```
plot(numbers, h)
```

затем выполнить следующие команды

```
figure()
plot(numbers, H)
```

Стандартно Python при построении графиков рисует прямые отрезки между точками со значениями. Можно также организовать другие способы вывода. Например, команда

```
plot(h, '*')
```

построит только точки значений отмеченные символом * (можно использовать также другие символы).

Кроме того, Python позволяет добавлять дополнительную информацию на графики. Например, можно добавить легенду с помощью следующей команды

```
legend('hH')
```

или название графика

```
title('График')
```

Команда

```
axis([xmin, xmax, ymin, ymax])
```

задает область построения графика: для оси x участок от `xmin` до `xmax`, для оси y участок от `ymin` до `ymax`

Сохранение графика в фалы соответствующих форматов можно выполнить с помощью следующих команд:

```
savefig('pic.png')
savefig('pic.jpg')
savefig('pic.gif')
savefig('pic.eps')
savefig('pic.pdf')
```

Матрицы. Для линейной алгебры могут понадобится стандартные операции, например, умножение матрицы на вектор. Для этого можно использовать специальный тип. Например, мы хотим вычислить вектор y следующим образом $\mathrm{y} = \mathrm{Ax}$, где A — матрица размера 2×2, x — вектор.

Это можно реализовать следующим образом (сценарий mat_vec_mult.py[12])

```
# -*- coding: utf-8 -*-

from numpy import zeros, mat, transpose

x = zeros(2)
x = mat(x)
x = transpose(x)
x[0] = 1.0; x[1] = 3.0

A = zeros((2,2))
A = mat(A)
A[0,0] = 1.0; A[0,1] = 2.0
A[1,0] = 3.0; A[1,1] = 4.0

y = A*x

print y
```

Здесь сначала `x` создан как объект типа `array`, затем тип переменной `x` преобразован в тип `matrix` с помощью команды `mat(x)`. Далее матрица-строка `x` преобразована в матрицу столбец с помощью операции транспонирования `transpose(x)`. Создана матрица `A` как массив размерности 2×2 командой `zeros((2,2))`, преобразована в тип `matrix`. Затем заданы значения каждому элементу матрицы. И наконец, умножение матрицы на вектор выполнено операцией `y=A*x`. Запуск сценария дает

```
[[  7.]
 [ 15.]]
```

Входные данные. Как известно для выполнения вычислений компьютерным программам необходимы входные данные. В предыдущих примерах мы определяли входные данные как переменные, задавая их значения в самой программе. Однако очень часто удобно вводить данные во время выполнения программы посредством диалога с пользователем. Ниже приведен пример, где сценарий задает вопрос, а пользователь должен ввести данные.

[12] `src-python/mat_vec_mult.py`

```
# -*- coding: utf-8 -*-

age = input('Введите Ваш возраст\n')
print u'Вы на половине пути к %d' % (age*2)
```

После запуска сценария первая строка определяет переменную age и устанавливает ей значение, введенное пользователем. Вторая строка объединяет вычисление и вывод результата на экран.

Функция input удобна для вводя чисел, списков и кортежей. Для ввода строковых переменных пользователь должен заключать в кавычки вводимые данные, либо можно воспользоваться функцией raw_input:

```
name = raw_input('Введите Ваше имя: ')
```

Конечно, существуют другие способы ввода входных данных, например, графический интерфейс пользователя, параметры командной строки или чтение входных данных из файла.

Символьные вычисления. Несмотря на то, что наш курс посвящен в основном численным методам, возникнут ситуации, когда будут полезны *символьные* (точные или аналитические) вычисления. Выполнение символьных вычислений означает, что мы осуществляем вычисления с символами, а не с числовыми значениями. Проиллюстрируем отличие символьных вычислений от численных на небольшом примере. Численные расчеты могут быть следующими

```
x = 2
y = 3
z = x*y
print z
```

В результате на экране мы получим число 6. Символьная версия приведенного кода может быть следующей

```
from sympy import *
x, y = symbols('x y')
```

```
z = x*y
print z
```

Этот код дает символьный результат x*y. Отметим, что в этом случае не задаются числовые значения символьным переменным, а используются только символы.

Выполнение символьных вычислений в Python обеспечивает пакет SymPy. Каждый символ представлен переменной, но имя символа должно быть определено либо командой `Symbol(имя)` для одного символа, либо командой `symbols(имя1 имя2 ...)` для нескольких символов. В сценарии symbolic.py[13]

```
# -*- coding: utf-8 -*-

from sympy import *

x = Symbol('x')
y = Symbol('y')

print x**2 + 3*y**3
print diff(x**3, x)              # дифференцирование
print integrate(tan(x), x)       # интегрирование
print simplify((x**2+x**3)/x**2) # упрощение выражения
print limit((1+x)**(1/x), x, 0)  # вычисление предела
print solve(x**2-16, x)          # решение уравнения x^2=16
```

Также доступны другие символьные вычисления: разложение в ряд Тейлора, линейная алгебра (операции с матрицами и векторами), решение некоторых дифференциальных уравнений и т.д.

Удаление не используемых данных

Python имеет *автоматический сборщик мусора*. Это означает, что нет необходимости удалять переменные (или объекты), которые больше не используются. Python заботится об этом сам. Это отличает его от, например, Matlab, где иногда требуется явно удалять переменные.

[13] `src-python/symbolic.py`

Совет

Если выражение в программе слишком длинное, его можно продолжить на следующей строке вставкой обратной косой черты перед переходом на новую строку. Однако, не должно быть пробелов после обратной косой черты.

1.2. Основные инструкции

1.2.1. Инструкция `if`, двоеточие и отступы

Рассмотрим следующий пример, являющийся ядром так называемого алгоритма блуждания, используемого во многих приложениях, например, в производство новых материалов и при изучения мозга. Действие заключается в случайном движении на север (N), юг (S), запад (W) или восток (E) с одинаковой вероятностью.

Нам нужно выбрать случайным образом одно из четырех чисел, чтобы задать направление движения. Таким образом, мы выбираем число, выполняем соответствующее движение, и повторяем этот процесс много раз. Итоговый путь является типичной реализацией диффузии молекулы.

Для реализации этого процесса мы сгенерируем случайное число из интервала $[0, 1)$. Будем считать, что $[0, 0.25)$ соответствует N, $[0.25, 0.5)$ — E, $[0.5, 0.75)$ — S и $[0.75, 1)$ — W. Пусть `x`, `y` — координаты точки на плоскости, а `d` — длина перемещения. Мы должны сначала проверить принадлежность `r` соответсвующему отрезку и выполнить действие. Когда ответ на вопрос *если* положительный (`true`), мы выполняем действие. Если ответ отрицательный (`false`), мы обрабатываем вопрос *иначе если* и т.д. Последний тест может обрабатываться как *иначе*, так как уже обработаны все варианты.

Код на Python для такой проверки выглядит следующим образом

```
# -*- coding: utf-8 -*-

import random

x = 0.0; y = 0.0; d = 0.1

r = random.random()
if 0 <= r < 0.25:
    # перемещаемся на север
    y = y + d
elif 0.25 <= r < 0.5:
    # перемещаемся на восток
    x = x + d
elif 0.5 <= r < 0.75:
    # перемещаемся на юг
    y = y - d
else:
    # перемещаемся на запад
    x = x - d

print x, y
```

Python предоставляет зарезервированные слова `if`, `elif` (сокращенно от *else if*) и `else`. Эти инструкции работают по следующим правилам:

- Инструкция работает следующим образом `if условие:`, `elif условие:`, `else:`, здесь `условие` — булевское выражение возвращающее значение `True` или `False`.
- Если `условие` равно `True`, следующий блок выражений с отступами и остальные `if`, `elif` и `else` пропускаются.
- Если `условие` равно `False` программа переходит к следующему `if`, `elif` или `else`.

Как видим здесь используются булевские выражения, которые могут использовать следующие *логические операторы*: `==`, `!=`, `<`, `<=`, `>` и `>=`.

1.2.2. Функции

Функции широко используются в программировании и являются одним из основных понятий, которые необходимо освоить. В простейшем случае понятие функция в про-

грамме очень близко к понятию математической функции: некоторое входное число x преобразуется в некоторое выходное число. Один из примеров функция $\mathrm{arctg}\, x$, которой в Python соответствует функция `atan(x)`. Функции в Python могут иметь ряд входных параметров и возвращать одно или несколько значений или ничего не возвращать. Цель использования функций состоит из двух частей:

- группировать строки кода, которые естественным образом связаны друг с другом и/или
- параметризовать набор выражений таких, которые можно написать только один раз, а затем использовать с разными параметрами

Приведем примеры использования функций в разных ситуациях.

Мы можем, например, модифицировать сценарий ball.py[14] из раздела (1.1.1), добавив функцию, как сделано в сценарии ball_func.py[15]:

```
# -*- coding: utf-8 -*-
# Программа для вычисления положения мяча при вертикальном движении
# с использованием функции

def y(t):
    v0 = 5       # Начальная скорость
    g = 9.81     # Ускорение свободного падения
    return v0*t - 0.5*g*t**2

t = 0.6      # Время
print y(t)
t = 0.9
print y(t)
```

При выполнении этого сценария Python интерпретирует код от строки с ключевым словом `def` до строки с ключевым словом `return` (обратите внимание на двоеточие и отступы) как определение функции с именем `y`. Выражение, содержащее ключевое слово `return` интерпретируется Python следующим образом: сначала выполняется вычисление, затем возвращается его результат, в том месте где функция будет вызвана. Функция зависит от `t`, т.е. от

[14] `src-python/ball.py`
[15] `src-python/ball_func.py`

одной переменной (мы будем говорить, что функция принимает один аргумент или входной параметр), значение которой должно быть задано при вызове функции.

Что на самом деле происходит, когда интерпретатором Python встречается такой код? Строка с ключевым словом `def` сообщает интерпретатору, что здесь определяется функция с именем `y`, которая имеет один входной параметр `t`.

Если функция содержит инструкции `if-elif-else`, каждый из блока может возвращать значение с помощью `return`:

```
def check_sign(x):
    if x > 0:
       return u'x --- положительное число'
    elif x < 0:
       return u'x --- отрицательное число'
    else:
       return u'x равно нулю'
```

В этой ситуации только один блок будет выполнен при вызове функции `check_sign`.

У функции могут отсутствовать аргументы, или быть много аргументов, которые просто перечисляются в круглых скобках, и разделены запятыми. Проиллюстрируем это на примере. Модифицируем пример с подбрасыванием мяча следующим образом. Предположим, что мы бросаем мяч не вертикально вверх, а под углом. В этом случае требуется две координаты для описания позиции мяча в каждый момент времени. Согласно закону Ньютона (сопротивлением воздуха пренебрегаем) вертикальная координата описывается формулой $y(t) = v_{0y}t - 0.5gt^2$, а горизонтальная координата — $x(t) = v_{0x}t$. Теперь можно включить оба эти выражения в новую версию сценария, который выводит позицию мяча для заданного момента времени. Допустим, мы хотим вычислить эти значения для двух моментов времени $t = 0.6$ с и $t = 0.9$ с. Мы можем задать значения компонент начальной скорости v_{0x} и v_{0y}. Сценарий может выглядеть следующим образом ball2d.py[16]

[16] `src-python/ball2d.py`

```
def y(v0y, t):
    g = 9.81
    return v0y*t - 0.5*g*t**2

def x(v0x, t):
    return v0x*t

initial_velocity_x = 2.0
initial_velocity_y = 5.0

time = 0.6
print x(initial_velocity_x, time), y(initial_velocity_x, time)
time = 0.9
print x(initial_velocity_x, time), y(initial_velocity_x, time)
```

В результате мы вычисляем и печатаем две координаты положения мяча для каждого из двух моментов времени. Отметим, что функции имеют два аргумента. Запуск сценария даст следующий вывод:

Terminal

```
1.2 -0.5658
1.8 -2.17305
```

Функция может не иметь возвращаемого значения (в этом случае инструкция `return` опускается), а также возвращать более одного значения. Например, две функции, которые мы только что определили, можно заменить одной

```
def xy(v0x, v0y, t):
    g = 9.81
    return v0x*t, v0y*t - 0.5*g*t**2
```

При этом возвращаемые значения разделяются запятой. При вызове функции аргументы должны следовать в том же порядке, как и в определении. Теперь мы можем напечатать результат следующим образом:

```
print xy(initial_velocity_x, initial_velocity_y, time)
```

Два возвращаемых значения могут быть присвоены двум переменным, например, так

```
x, y = xy(initial_velocity_x, initial_velocity_y, time)
```

Теперь переменные x и y могут использоваться в коде.

Существует возможность варьировать число входных и выходных параметров (используя *args и **kwargs конструкции для аргументов). Однако, мы не будем рассматривать этот вариант.

Переменные, которые определены внутри функции (например, g в xy), являются *локальными переменными*. Это означает, что они видимы только внутри функции. Поэтому, если мы случайно попытаемся использовать переменную g вне функции, мы получим сообщение об ошибке. Переменная time определена вне функции и поэтому является *глобальной переменной*. Она видима как вне, так и внутри функции(ий). Если мы определим одну глобальную и одну локальную переменные с одним и тем же именем, то внутри функции будет видима только локальная переменная, при этом глобальная переменная не изменяется при изменении локальной переменной с тем же именем.

Аргументы перечисленные в заголовке определения функции, как правило, являются локальными переменными. Если нужно изменить значение глобальной переменной внутри функции, следует определить переменную внутри функции как глобальную, т.е., если глобальная переменная имеет имя x, то нам нужно написать global x внутри определения функции, прежде чем изменять её значение. После выполнения функции, x будет иметь измененное значение. Следует стараться определять переменные там, где они необходимы.

Еще один полезный способ управления параметрами в Python заключается в использовании *именованных аргументов*. Этот подход позволяет задавать аргументам значения по умолчанию и дает больше свободы в вызове функций, так как порядок и количество аргументов может варьироваться.

Проиллюстрируем использование именованных аргументов на примере функции, аналогичной xy. Определим функцию xy_named следующим образом:

```
# -*- 4th Start
def xy_named(t, v0x = 0, v0y = 0):
    g = 9.81
    return v0x*t, v0y*t - 0.5*g*t**2
```

Здесь `t` — обычный или *позиционный аргумент*, тогда как `v0x` и `v0y` — *именованные аргументы*. В общем случае, может быть несколько позиционных и несколько именованных аргументов, но позиционные аргументы *всегда* должны следовать перед именованными в определении функции. Именованные аргументы имеют значение по умолчанию, в нашем примере `vx0` и `v0y` по умолчанию равны нулю. В сценарии функция `xy_named` может быть вызвана разными способами. Например,

```
print xy_named(0.6)
```

выполнит вычисления с `t = 0.6` и значениями,заданными по умолчанию (в нашем случае 0) для `v0x` и `v0y`. Два возвращаемых функцией `xy_named` значения будут выведены на экран. Если мы хотим использовать другое значение для переменной `v0y`, мы можем, например, написать

```
print xy_named(0.6, v0y=4.0)
```

т.е. выполнить функцию `xy_named` с `t = 0.6`, `v0x = 0` (значение по умолчанию) и `v0y = 4.0`. В случае, когда есть несколько позиционных аргументов, они должны следовать в том порядке, в котором перечислены в определении функции, если мы не задаем явно имя переменной в вызове функции. Используя явное задание имени переменной в вызове функции, мы можем использовать любой порядок аргументов функции. Например, мы можем вызвать функцию `xy_named` следующим образом:

```
print xy_named(v0y=4.0, v0x = 1.0, t = 0.6)
```

При использовании любого языка программирования существует хорошая традиция включать небольшое пояснение о том, что делает функция, если это не очевид-

но. Такое пояснение называется *строкой документации* (*doc string*), которую в Python следует помещать в начале функции. Это пояснение предназначено для программистов, которые желают понять код, так что следует описать цель данного кода и, возможно, описать аргументы и возвращаемые значения. Например, мы можем написать

```
# -*- 5th Start
def xy_named(t, v0x = 0, v0y = 0):
    """Вычисляются координаты x и y положения мяча в момент времени t """
    g = 9.81
    return v0x*t, v0y*t - 0.5*g*t**2
```

Следует отметить, что функция может быть вызвана из других функций, а также входными параметрами функции могут быть не только числа. Любые объекты могут быть аргументами функции, например, строковые переменные или функции.

Функции прямо передаются как аргументы других функций, что проиллюстрировано в сценарии functions_as_args.py[17]

```
# -*- coding: utf-8 -*-

def sum_xy(x, y):
    return x + y

def prod_xy(x, y):
    return x*y

def treat_xy(f, x, y):
    return f(x,y)

x = 2; y = 3
print treat_xy(sum_xy, x, y)
print treat_xy(prod_xy, x, y)
```

При запуске этот сценарий сначала выведет на экран сумму x и y, а затем произведение. Видно, что treat_xy принимает имя функции в качестве первого аргумента. Внутри treat_xy это имя используется для вызова соответствующей функции.

[17] src-python/functions_as_args.py

Функция также может быть определена внутри другой функции. В этом случае она становится *локальной* или *вложенной функцией*, видимой только функцией, внутри которой она определена. Функции определенные в главном сценарии называются *глобальными функциями*. Вложенная функция имеет полный доступ ко всем переменным *родительской функции*, т.е. функции, внутри которой она определена.

Короткие функции могут определяться компактно с помощью *лямбда функций*:

```
f = lambda x, y: x + 2*y
```

эквивалентно

```
def f(x, y):
    return x + 2*y
```

Синтаксис состоит из ключевого слова `lambda` и следующих за ним набора аргументов, двоеточия и некоторого выражения, дающим в результате объект, возвращаемый функцией. Лямбда функции особенно удобно использовать в качестве аргументов функций:

```
print treat_xy(lambda x, y: x*y, x, y)
```

Издержки при вызове функций

Вызов функций имеет недостаток, заключающийся в замедлении выполнения программы. Как правило, разбиение программы на функции считается хорошим тоном, но в частях, содержащий очень интенсивные вычисления, например, внутри длинных циклов, нужно находить баланс между удобством вызова функции и вычислительной эффективностью, избегающей вызовы функций. Можно предложить правило, когда разрабатывается программа, содержащая много функций, а затем на стадии оптимизации, когда

все вычисления корректны, удалять вызовы функций, которые замедляют выполнение кода.

Ниже приведен небольшой пример в оболочке IPython, где вычисляется процессорное время при выполнение вычислений с массивами при использовании и без использования вспомогательной функции

```
In [1]: import numpy as np

In [2]: a = np.zeros(1000000)

In [3]: def add(a, b):
   ...:     return a+b
   ...:

In [4]: %timeit for i in range (len(a)): a[i] = add(i, i+1)
   10 loops, best of 3: 150 ms per loop

In [5]: %timeit for i in range (len(a)): a[i] = i + i+1
  10 loops, best of 3: 93.2 ms per loop
```

1.2.3. Циклы

Многие вычисления являются повторяющимися и, естественно, языки программирования имеют некоторые циклические конструкции. В этом разделе мы рассмотрим такие конструкции языка Python.

Цикл `for`. Начнем с инструкции `for`. Предположим, мы хотим вычислить квадраты чисел от 3 до 7. Это можно сделать с помощью следующей инструкции:

```
for i in [3, 4, 5, 6, 7]:
    print i**2
```

Предупреждение

Обратите внимание на двоеточие и отступ!

Что происходит, когда интерпретатор Python обрабатывает данный код? Во первых ключевое слово `for` сообщает интерпретатору, что будет выполняться цикл. Python затем придерживается правил, определенных для таких конструкций, и понимает, что (в представленном примере) цикл должен быть выполнен 5 раз (т.е., следует выполнить 5 итераций), при этом переменная `i` принимает последовательно значения `3, 4, 5, 6, 7`. Во время каждой итерации выполняется выражение внутри цикла (т.е. `print i**2`). После каждой итерации, значение `i` автоматически изменяется. При достижении последнего значения, выполняется последняя итерация и цикл завершается. При выполнении сценарий выведет последовательно `9, 16, 25, 36` и `49`. Переменная `i` часто называется *счетчик цикла*, а выбор имени этой переменной (здесь `i`) остается за программистом.

Отметим, что внутри цикла может быть несколько выражений, которые будут выполнятся с одним и тем же значением `i`.

В языке Python целые значения, определяемые для счетчика цикла, часто получаются с помощью встроенной функции `range`. Функция `range` может быть вызвана разными способами, которые явно или неявно задают начальное, конечное значения и шаг изменения счетчика цикла. В общем случае, вызов выглядит следующим образом:

```
range(start, stop, step)
```

Такой вызов генерирует массив целых чисел от (включая) `start` до (не включая!) `stop` с шагом `step`, т.е. `stop-1` — последнее целое значение. С использованием функции `range` предыдущий пример будет выглядеть так:

```
for i in range(3, 8, 1):
    print i**2
```

По умолчанию, если функция `range` вызывается с двумя параметрами, они будут приняты в качестве `start` и `stop`, при этом `step=1`. Если задан только один аргумент, он используется в качестве `stop`. При этом шаг по умолчанию

равен 1, а значение start равно 0. Таким образом, следующий вызов

```
range(6)
```

вернет целые числа 0, 1, 2, 3, 4, 5.

Отметим, что можно сгенерировать убывающую последовательность целых чисел, задав start > stop и отрицательный шаг.

Модифициурем сценарий ball_plot.py[18] из раздела 1.1.3 для иллюстрации использования цикла for. В том примере мы вычисляли высоту мяча в каждую миллисекунду первой секунды его полета и строили график зависимости высоты от времени.

Предположим, мы хотим найти максимальную высоту в течение этого времени. Один из вариантов реализации этого может быть следующим: вычисляем все тысяча значений высоты, сохраняем их в массив, а затем пробегаем весь массив, чтобы найти максимальное значение. Сценарий (max_height.py[19]) может выглядеть так:

```
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt

v0 = 5.0
g = 9.81
t = np.linspace(0, 1, 1000)
y = v0*t - 0.5*g*t**2

max_height = y[0]
for i in range(1, 1000):
    if y[i] > max_height:
        max_height = y[i]

print u'Максимальная достигнутая высота равна %f м' % (max_height)

plt.plot(t, y)
plt.xlabel(u'Время (с)')
plt.ylabel(u'Высота (м)')
plt.show()
```

[18] src-python/ball_plot.py

[19] src-python/max_height.py

Запуск программы даст

Terminal

```
Максимальная достигнутая высота равна 1.274210 м
```

что хорошо согласуется с построенным графиком.

Иногда используются *вложенные циклы*, например в линейной алгебре. Скажем, нам нужно найти максимум среди элементов матрицы `A` размера 4×4. Фрагмент кода может выглядеть так:

```
max_number = A[0][0]

for i in range(4):
    for j in range(4):
        if A[i][j] > max_number:
        max_number = A[i][j]
```

Следует отметить, что имя счетчика для каждого цикла из вложенных должно быть единственным. Кроме того, каждый вложенный цикл может содержать несколько строк кода как до, так и после следующего внутреннего цикла.

Векторизованные вычисления, которые мы использовали в ball_plot.py[20] из раздела 1.1.3 могли быть заменены обходом массива `t` и выполнением вычислений высоты по заданной формуле для каждого элемента `t`. Однако, следует знать, что векторизованные вычисления выполняются гораздо быстрее.

Цикл `while`. В языке Python имеется еще одна стандартная циклическая конструкция — цикл `while`. Для иллюстрации использования этого цикла рассмотрим другую модификацию сценария ball_plot.py[21] из раздела. Теперь мы изменим его так, чтобы сценарий находил время полета мяча. Предположим, что мы подбросили мяч с немного меньшей начальной скоростью 4.5 м/с. Так как мы все еще будем рассматривать первую секунду полета, то высота в конце этого интервала будет отрицательной. Это означает, что мяч упал ниже своего начального положе-

[20] `src-python/ball_plot.py`
[21] `src-python/ball_plot.py`

ния. В нашем массиве y мы будем иметь ряд отрицательных значений, которые расположены в конце массива. В сценарии ball_time.py[22] находится момент времени, когда значение высоты становится отрицательным, т.е., когда мяч пересекается прямую $y = 0$. Сценарий может быть следующим:

```
# -*- coding: utf-8 -*-

import numpy as np

v0 = 4.5
g = 9.81
t = np.linspace(0, 1, 1000)
y = v0*t - 0.5*g*t**2

i = 0
while y[i] >= 0:
    i += 1

print u'y = 0 в момент времени ', 0.5*(t[i-1] + t[i])

import matplotlib.pyplot as plt

plt.plot(t, y)
plt.xlabel(u'Время (с)')
plt.ylabel(u'Высота (м)')
plt.show()
```

При выполнении сценария получим

Terminal

```
y = 0 в момент времени  0.917417417417
```

В приведенном примере цикл `while` выполняется до тех пор, пока булевское выражение `y[i] > 0` возвращает значение `True`. Отметим, что в этом случае счетчик цикла `i` введен и инициализирован (`i = 0`) до начала выполнения цикла и обновляется (`i += 1`) внутри цикла. Таким образом, для каждой итерации `i` явно увеличивается на 1.

В отличие от цикла `for`, программист не должен определять количество итераций при использовании цикла `while`. Он просто выполняется пока булевское выражение не вернет значение `False`. Таким образом, в этом случае

[22] `src-python/ball_time.py`

счетчик цикла не обязателен. Кроме того, если в цикле `while` используется счетчик, то он не увеличивается автоматически, это нужно делать явно. Конечно, как и в цикле `for` и в инструкции `if` может быть несколько строк кода в теле цикла `while`. Любой цикл `for` может быть реализован с помощью `while`, но циклы `while` являются более общими и не все из них можно реализовать с помощью `for`.

Следует быть осторожным с так называемыми *бесконечными циклами*. Могут возникнуть (непреднамеренно) случаи, когда тест в инструкции `while` никогда не вернет значение `False`, и сценарий не сможет выйти из цикла. Если вы случайно зациклите сценарий то можно нажать комбинацию клавиш `Ctrl-C`, чтобы остановить работу сценария.

1.2.4. Списки и кортежи

Списки. Как мы видели ранее набор чисел может хранится в массивах, которые мы можем обрабатывать целиком и поэлементно. В языке Python существует другой способ объединения данных, которые могут широко использоваться, как минимум не в вычислительных процедурах. К таким конструкциям относятся *списки*.

Списки очень похожи на массивы, но есть плюсы и минусы, которые стоит рассмотреть. Например, количество элементов списка может меняться, в то время как массивы имеют фиксированную длину, которую необходимо знать в момент выделения памяти. Элементы списка могут быть разных типов, в то время как элементы массива должны быть одного и того же типа. В общем случае, списки предоставляют большую гибкость, чем массивы. С другой стороны, массивы дают большую скорость вычислений, чем списки, что делает выбор массивов предпочтительным, если нет необходимости в гибкости, предоставляемой списками. Массивы также требуют меньше памяти и существует большое число готовых программ для различных математических вычислений. Векторизованные вычисления требуют использования массивов.

Функция range на самом деле возвращает список. К элементу списка можно обращаться как к элементу массива x[i]. Как и для массивов индексы элементов списка пробегают значения от 0 до n-1, если n — количество элементов списка. Можно преобразовать список в массив следующим образом: x = array(L).

Список можно создать, например, так:

```
x = [’hello’, 3, 3.15, 4]
```

В этом случае, x[0] содержит строку hello, x[1] содержит целое число 3, x[2] содержит действительное (float) число 3.15 и т.д. Мы можем добавлять и удалять элементы массива, как показано в следующем примере

```
x = [’hello’, 3, 3.15, 4]
x.insert(0, -2)    # x становится [-2, ’hello’, 3, 3.15, 4]
del x[3]           # x становится [-2, ’hello’, 3, 4]
x.append(3.15)     # x становится [-2, ’hello’, 3, 4, 3.15]
```

Метод append добавляет элемент в конец списка. Если необходимо, можно создать пустой список x = [] а потом в цикле добавлять элементы к списку. Чтобы узнать длину списка, можно воспользоваться функцией len(x). Эта функция полезна, когда нам нужно обойти список по индексам, так как функция range(len(x)) возвращает все допустимые индексы.

Ранее мы видели обходить все элементы массива с помощью цикла for. Если нам нужно пробежать все элементы списка, мы можем сделать это, как показано в следующем примере:

```
x = [’hello’, 3, 3.15, 4]
for e in x:
    print ’Элемент x: ’, e
print ‘Это были все элементы списка x‘
```

Как видно мы пробегаем элементы массива без использования индексов. Следует понимать, что, когда мы так используем цикл, мы не можем изменять значения элемента списка x, изменяя e. Это означает, что, запись e +=

2 ничего не изменит в списке x, так как e используется только для чтения элементов списка.

В языке Python существует специальная конструкция, позволяющая пробежать все элементы списка, выполнить операцию с каждым элементом и сохранить новые элементы в другой список. Назовем эту конструкцию *охват списка* и проиллюстрируем примером:

```
List1 = [1, 2, 3, 4]
List2 = [e*10 for e in List1]
```

Этот фрагмент кода создает новый список List2, содержащий элементы 10, 20, 30 и 40. Выражение в квадратных скобках for e in List1 означает, что будет последовательно пробегать все элементы списка List1, и для каждого e будет создан новый элемент списка List2 со значением e*10. В более общем случае:

```
List2 = [E(e) for e in List1]
```

где E(e) означает любое выражение содержащее e.

В некоторых случаях требуется одновременно пробегать 2 или более списков. Python имеет удобную для этих целей функцию zip. Пример использования функции zip будет дан ниже в сценарии file_handle.py[23].

Кортежи. Также кратко упомянем *кортежи*, конструкции очень похожие на списки. Основное их отличие заключается в том, что кортежи не могут быть изменены. Новичкам может показаться странным, что такие ”константные списки” могут быть даже предпочтительнее списков. Однако, свойство постоянности — хорошая мера предосторожности от непреднамеренных изменений. Кроме того, в Python доступ к данным в кортежах быстрее, чем в списках, что способствует более быстрому коду. На основе примера, который был дан выше, мы можем создать кортеж и вывести на экран его содержимое:

```
x = (’hello’, 3, 3.15, 4)
for e in x:
```

[23] src-python/file_handle.py

```
    print 'Элемент x: ', e
print 'Это были все элементы списка x'
```

Попытка использовать `insert` или `append` к кортежу выдаст сообщение об ошибке, гласящую что объект типа кортеж не имеет таких атрибутов.

1.2.5. Чтение из файлов и запись в файлы

Входные данные для программы часто можно получить из файла, а результаты вычислений часто нужно записывать в файл. Чтобы проиллюстрировать простейшую работу с файлами, рассмотрим пример, где читаются координаты x и y из файла, содержащего две колонки, применяется функция f к y, и записывается результат в новый файл с двумя колонками. Будем считать, что первая строка файла с данными — это заголовок файла, который будем пропускать:

```
# координат x и y
1.0  3.0
2.0  3.9
3.0  5.3
4.0  6.0
```

Соответствующий сценарий на Python дан в файле file_handle.py[24]

```
# -*- coding: utf-8 -*-

filename = 'data.txt'
infile = open(filename, 'r')   # открытие файла для чтения
line = infile.readline()       # чтение первой строки

# Чтение x и y из файла и сохранение их в списки
x = []; y = []
for line in infile:
    words = line.split()       # разбиение строки на слова
    x.append(float(words[0]))
    y.append(float(words[1]))
infile.close()
```

[24] src-python/file_handle.py

```
# Преобразование координаты y
from math import log

def f(y):
    return log(y)

for i in range(len(y)):
    y[i] = f(y[i])

# Запись x и y в файл из двух колонок
filename = 'tmp.txt'
outfile = open(filename, 'w')   # открытие файла для записи
outfile.write('# координаты x и y\n')
for xi, yi in zip(x, y):
    outfile.write('%10.5f %10.5f\n' % (xi, yi))
outfile.close()
```

Такие файлы, содержащие строку комментариев и колонки чисел, достаточно часто используются при научных вычислениях. Поэтому в numpy реализован функционал облегчающий чтение и запись в такие файлы. Ниже представлен пример, реализующий ту же функциональность, что и предыдущий, с использованием функций `loadtxt` и `savetxt` из numpy file_handle_np.py[25]:

```
# -*- coding: utf-8 -*-

filename = 'data.txt'
import numpy as np
data = np.loadtxt(filename, comments="#")
x = data[:,0]
y = data[:,1]
data[:,1] = np.log(y)
filename = 'tmp.txt'
outfile = open(filename, 'w')
outfile.write('# координаты x и y\n')
np.savetxt(outfile, data, fmt='%10.5f')
```

[25] src-python/file_handle_np.py

Решение систем линейных уравнений

2

Проблема решения линейной системы

$$Ax = b \tag{2.1}$$

является центральной в научных вычислениях. В этой главе мы остановимся на методах решения систем вида (2.1). Сначала остановимся на методе исключения Гаусса, а затем рассмотрим некоторые итерационные методы.

2.1. Прямые методы линейной алгебры

Одной из основных задач вычислительной математики является проблема решения систем линейных алгебраических уравнений с вещественными ко- эффициентами. Для нахождения приближенного решения систем уравнений используются прямые и итерационные методы. Математический аппарат ли- нейной алгебры базируется на понятиях нормы вектора и матрицы, числа обусловленности. Рассматриваются классические методы исключения неиз- вестных, отмечаются особенности решения задач с симметричной веществен- ной матрицей.

2.1.1. Метод исключения Гаусса

Начнем с обсуждения того, как можно легко решать треугольные системы. Затем опишем приведение системы общего вида к треугольной форме при помощи преобразований Гаусса. И, наконец, учитывая то, что полученный метод ведет себя очень плохо на нетривиальном классе задач, рассмотрим концепцию выбора ведущих элементов.

Треугольные системы. Рассмотрим следующую треугольную 2×2-систему:

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Если $l_{11}, l_{22} \neq 0$, то неизвестные могут быть определены последовательно:

$$\begin{cases} x_1 &= b_1/l_{11}, \\ x_2 &= (b_2 - l_{21}x_1)/l_{22} \end{cases}$$

Это 2×2-версия алгоритма, известного как *прямая подстановка*. Общую процедуру получаем, разрешая i-е уравнение системы $Lx = b$ относительно x_i:

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii}.$$

Если вычисления выполнить для i от 1 до n, то будут получены все компоненты вектора x. Заметим, что на i-м шаге необходимо скалярное произведение векторов $L(i, 1 : i-1)$ и $x(1 : i-1)$. Так как b_i содержится только в формуле для x_i, мы можем записать x_i на месте b_i.

Прямая подстановка

Предположим, что $L \in \mathbb{R}^{n \times n}$ — нижняя треугольная матрица и $b \in \mathbb{R}^n$. Следующий код Python заменяет b на решение системы $Lx = b$. Матрица L должна быть невырождена.

```
b[0] = b[0]/L[0,0]
for i in range(1,len(b)):
    b[i] = (b[i] - np.dot(L[i,:i], b[:i]))/L[i,i]
```

Аналогичный алгоритм для верхней треугольной системы $Ux = b$ называется *обратная подстановка*. Вот формула для x_i:

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}.$$

и снова x_i можно записать на месте b_i.

Обратная подстановка

Если матрица $U \in \mathbb{R}^{n \times n}$ верхняя треугольная и $b \in \mathbb{R}^n$, то следующий код Python заменяет b на решение системы $Ux = b$. Матрица U должна быть невырождена.

```
b[-1] = b[-1]/U[-1,-1]
for i in range(len(b)-2, -1, -1):
    b[i] = (b[i] - np.dot(U[i,i+1:], b[i+1:]))/U[i,i]
```

Отметим, что при реализации формул прямой и обратной подстановки мы использовали срезы массивов (см. раздел 1.1.4). В первом алгоритме `L[i,:i]` означает, что берется из строки двумерного массива с индексом `i` все элементы с нулевого до `i-1`-го включительно, а `b[:i]` — элементы массива `b` с индексами от `0` до `i-1` включительно. Во втором алгоритме используются срезы `U[i,i+1:]`, содержащий от `i+1`-го до последнего (включительно) элементы `i`-той строки, и `b[i+1:]` с элементами от `i+1`-го до последнего (включительно). Кроме того использовалась функция `dot` модуля `numpy`, которая вычисляет скалярное произведение двух векторов. Таким образом, мы здесь использовали векторизованные вычисления.

LU-разложение. Как мы только что видели, треугольные системы решаются «легко». Идея метода Гаусса — это преобразование системы (2.1) в эквивалентную треуголь-

ную систему. Преобразование достигается соответствующих линейных комбинаций уравнений. Например, в системе

$$3x_1 + 5x_2 = 9,$$
$$6x_1 + 7x_2 = 4,$$

умножая ее первую строку на 2 и вычитая ее из второй части, мы получим

$$3x_1 + 5x_2 = 9,$$
$$-3x_2 = -14.$$

Это и есть метод исключений Гаусса при $n = 2$. Дадим полное описание этой важной процедуры, причем опишем ее выполнение на языке матричных разложений. Данный пример показывает, что алгоритм вычисляет нижнюю треугольную матрицу L и верхнюю треугольную матрицу U так, что $A = LU$, т.е.

$$\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix}$$

Решение исходной задачи $Ax = b$ находится посредством последовательного решения двух треугольных систем:

$$Ly = b, \quad Ux = y \quad \Rightarrow Ax = LUx = Ly = b$$

Матрица преобразования Гаусса. Чтобы получить разложение, описывающее исключение Гаусса, нам нужно иметь некоторое матричное описание процесса обнуления матрицы. Пусть $n = 2$, тогда как $x_1 \neq 0$ и $\tau = x_2/x_1$, то

$$\begin{bmatrix} 1 & 0 \\ -\tau & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}$$

В общем случае предположим, что $x \in \mathbb{R}^n$ и $x_k \neq 0$. Если

$$\tau^{(k)T} = [\underbrace{0, \ldots, 0}_{k}, \tau_{k+1}, \ldots, \tau_n], \quad \tau_i = \frac{x_i}{x_k} \quad i = k+1, k+2, \ldots, n$$

и мы обозначим

$$M_k = I - \tau^{(k)} e_k^T, \tag{2.2}$$

где

$$e_k^T = [\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{n-k}],$$
$$I = [e_1, e_2 \ldots, e_n]$$

то

$$M_k x = \begin{bmatrix} 1 \ldots & 0 & 0 \ldots 0 \\ \vdots \ddots & \vdots & \vdots \ddots \vdots \\ 0 \ldots & 1 & 0 \ldots 0 \\ 0 \ldots & -\tau_{k+1} & 1 \ldots 0 \\ \vdots \ddots & \vdots & \vdots \ddots \vdots \\ 0 \ldots & -\tau_n & 0 \ldots 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Матрица M_k — это матрица *преобразования Гаусса*. Она является нижней унитреугольной. Компоненты $\tau_{k+1}, \tau_{k+2}, \ldots, \tau_n$ — это *множители Гаусса*. Вектор $\tau^{(k)}$ называется *вектором Гаусса*.

Для реализации данных идей имеется функция, которая вычисляет вектор множителей. Если `x` — массив из `n` элементов и `x[0]` ненулевой, функция `gauss` возвращает вектор длины $n-1$, такой, что если `M` — матрица преобразования Гаусса, причем `M[1:,1] = -gauss(x)` и `y = dot(M,x)`, то `y[1:] = 0`:

```
def gauss(x):
    x = np.array(x, float)
    return x[1:]/x[0]
```

Применение матриц преобразовния Гаусса. Умножение на матрицу преобразования Гаусса выполняется достаточно просто. Если матрица $C \in \mathbb{R}^{n \times r}$ и $M_k = I - \tau^{(k)} e_k^T$, тогда преобразование вида

$$M_k C = (I - \tau^{(k)} e_k^T) C = C - \tau^{(k)} (e_k^T C)$$

осуществляет одноранговую модификацию. Кроме того, поскольку элементы вектора $\tau^{(k)}$ равны нулю от первого до k-го равны нулю, то в каждой k-ой строке матрицы C задействованы лишь элементы, начиная с $k+1$-го. Следова-

тельно, если “C“ — двумерный массив, задающий матрицу C, и “M“ задает $n \times n$-преобразование Гаусса M_1, причем “M[1:,1] = -t“, “t“ — множитель Гаусса, соответствующий $\tau^{(1)T}$, тогда следующая функция заменяет C на $M_1 C$:

```
def gauss_app(C, t):
    C = np.array(C, float)
    t = np.array([[t[i]] for i in range(len(t))], float)
    C[1:, :] = C[1:, :] - t*C[0, :]
    return C
```

Отметим, что если матрица `M[k+1:,k] = -t`, тогда обращение вида `C[k:,:] = gauss_app(C[k:,:], t)` заменяет C на $M_k C$

Матрицы преобразовния Гаусса $M_1, M_2, \ldots, M_{n-1}$, как правило, можно подобрать так, что матрица $M_{n-1} M_{n-2} \ldots M_1 A = U$ является верхней треугольной. Легко убедиться, что если $M_k = I - \tau^{(k)} e_k^T$, тогда обратная к ней задается следующим выражением $M_k^{-1} = I + \tau^{(k)} e_k^T$ и поэтому

$$A = LU, \tag{2.3}$$

где

$$L = M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1}.$$

Очевидно, что L — это нижняя унитреугольная матрица. Разложение (2.3) называется *LU-разложением* матрицы A. Необходимо проверять *ведущие элементы* матрицы A (a_{kk}) на нуль, чтобы избежать деления на нуль в функции `gauss`. Это говорит о том, что LU-разложение может не существовать. Известно, что LU-разложение матрицы A существует, если главные миноры матрицы A не равны нулю при этом оно единственно и $\det A = u_{11} u_{22} \cdots u_{nn}$.

Реализация. Рассмотрим пример при $n = 3$:

```
In [1]: import numpy as np

In [2]: A = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 10]])

In [3]: A
Out[3]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6, 10]])
```

```
In [4]: M1 = np.array([[1, 0, 0], [-2, 1, 0], [-3, 0, 1]])

In [5]: M1
Out[5]:
array([[ 1,  0,  0],
       [-2,  1,  0],
       [-3,  0,  1]])

In [6]: np.dot(M1, A)
Out[6]:
array([[  1,   4,   7],
       [  0,  -3,  -6],
       [  0,  -6, -11]])

In [7]: M2 = np.array([[1, 0, 0], [0, 1, 0], [0, -2, 1]])

In [8]: M2
Out[8]:
array([[ 1,  0,  0],
       [ 0,  1,  0],
       [ 0, -2,  1]])

In [9]: np.dot(M2,np.dot(M1,A))
Out[9]:
array([[ 1,  4,  7],
       [ 0, -3, -6],
       [ 0,  0,  1]])
```

Функция `numpy.dot`

Обратите внимание, что в приведенном примере мы использовали функцию `dot` модуля `numpy`, которая выполняет умножение матриц в "правильном смысле", в то время как выражение `M1*A` производит поэлементное умножение.

Обобщение этого примера позволяет представить k-й шаг следующим образом:

- Мы имеем дело с матрицей $A^{(k-1)} = M_{k-1} \cdots M_1 A$, которая с 1-го по $(k-1)$-й столбец является верхней треугольной.
- Поскольку мы уже получили нули в столбцах с 1-го по $(k-1)$-й, то преобразование Гаусса можно применять только к столбцам с k-го до n-го. На самом деле нет

необходимости применять преобразование Гаусса также и k-му столбцу, так как мы знаем результат.
- Множители Гаусса, задающие матрицу M_k получаются по матрице $A(k:n,k)$ и могут храниться в позициях, в которых получены нули.

С учетом сказанного выше мы можем написать следующую функцию:

```
def lu(A):
    LU = np.array(A, float)
    for k in range(LU.shape[0]-1):
        t = gauss(LU[k:, k])
        LU[k+1:, k] = t
        LU[k:, k+1:] = gauss_app(LU[k:, k+1:], t)

    return LU
```

Эта функция возвращает LU-разложение матрицы A. Где же храниться матрица L? Дело в том, что если $L = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}$, то элементы с $(k+1)$-го до n-го в k-том столбце матрицы L равны множителям Гаусса $\tau_{k+1}, \tau_{k+2}, \dots, \tau_n$ соответственно. Этот факт очевиден, если посмотреть на произведение, задающее матрицу L:

$$L = (I + \tau^{(1)}e_1^T \cdots (I + \tau^{(n-1)}e_{n-1}^T)) = I + \sum_{k=1}^{n-1} \tau^{(k)}e_k^T.$$

Поэтому элементы $l_{ik} = lu_{ik}$ для всех $i > k$. Здесь lu_{ik} — элементы матрицы возвращаемой функцией `lu`.

После разложения матрицы A с помощью функции `lu` в возвращаемом массивы будут храниться матрицы L и U. Поэтому мы можем решить систему $Ax = b$, используя прямую и обратную подстановки описанные в разделе 2.1.1:

```
def solve_lu(A, b):
    LU = lu(A)
    b = np.array(b, float)
    for i in range(1, len(b)):
        b[i] = b[i] - np.dot(LU[i, :i], b[:i])
    for i in range(len(b)-1, -1, -1):
        b[i] = (b[i] - np.dot(LU[i, i+1:], b[i+1:]))/LU[i, i]
    return b
```

Замечание

Отметим, что во всех представленных функциях мы выполняли явное преобразование входных параметров в массивы NumPy с элементами типа `float`. Это позволит правильно работать функциям в случае, если мы по ошибке создадим входные параметры не как массивы, а как списки.

Тестирование. Как известно метод Гаусса является прямым, т.е. дает точное решение системы линейных уравнений. Для проверки реализации решения системы линейных уравнений методом Гаусса мы можем написать следующую функцию:

```
def test_solve_lu():
    A = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 10]])
    expected = np.array([-1./3, 1./3, 0])
    b = np.dot(A, expected)
    computed = solve_lu(A, b)
    tol = 1e-14
    success = np.linalg.norm(computed - expected) < tol
    msg = 'x_exact = ' + str(expected) + '; x_computed = ' + str(computed)
    assert success, msg
```

Замечание

Здесь мы задали матрицу `A` системы и точное решение `expected`, на основе которых получили вектор правой части `b = np.dot(A,x)`. Для сравнения численного решения с точным используется функция `np.linalg.norm`. В случае вызова с одним аргументом вычисляется l_2-норма: $\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$.

Выбор ведущего элемента. Как уже упоминалось, LU-разложение может не существовать. В методе Гаусса с выбором ведущего элемента на очередном шаге исключается неизвестное, при котором коэффициент по модулю

является наибольшим. В этом случае метод Гаусса применим для любых невырожденных матриц ($\det A \neq 0$).

Такая стратегия предполагает переупорядочивание данных в виде перестановки двух матричных строк. Для этого используются понятие перестановочной матрицы. *Перестановочная матрица* (или *матрица перестановок*) — это матрица, отличающаяся от единичной лишь перестановкой строк, например

$$P = \begin{bmatrix} 0\,0\,0\,1 \\ 1\,0\,0\,0 \\ 0\,0\,1\,0 \\ 0\,1\,0\,0 \end{bmatrix}.$$

Перестановочную матрицу нет необходимости хранить полностью. Гораздо более эффективно перестановочную матрицу можно представить в виде целочисленного вектора p длины n. Один из возможных способов такого представления — это держать в p_k индекс столбца в k-й строке, содержащий единственный элемент равный 1. Так вектор $p = [4, 1, 3, 2]$ соответствует кодировке приведенной выше матрицы P. Также возможно закодировать P указанием индекса строки в k-ом столбце, содержащего 1, например, $p = [2, 4, 3, 1]$.

Если P — это матрица перестановок, а A — некоторая матрица, тогда матрица AP является вариантом матрицы A с переставленными столбцами, а PA — вариантом матрицы A с переставленными строками.

Перестановочные матрицы ортогональны, и поэтому если P — перестановочная матрица, то $P^{-1} = P^T$.

В этом разделе особый интерес представляют *взаимные перестановки*. Такие перестановки осуществляют матрицы, получаемые простой переменой мест двух строк единичной матрицы, например

$$E = \begin{bmatrix} 0\,0\,0\,1 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 1\,0\,0\,0 \end{bmatrix}.$$

Взаимные перестановки могут использоваться для описания перестановок строк и столбцов матрицы. В приведенном примере порядка 4×4 матрица EA отличается от матрицы A перестановкой 1-й и 4-й строк. Аналогично матрица AE отличается от матрицы A перестановкой 1-го и 4-го столбцов.

Если $P = E_n E_{n-1} \cdots E_1$ и каждая матрица E_k является единичной с переставленными k-й и p_k-й строками, то вектор $p = [p_1, p_2, \ldots, p_n]$ содержит всю необходимую информацию о матрице P. Действительно, вектор x может быть замещен на вектор Px следующим образом:

$$\textbf{for } k = 1 : n$$
$$x_k \leftrightarrow x_{p_k}$$

Здесь символ $\leftrightarrow$ обозначает «выполнение перестановки»:

$$x_k \leftrightarrow x_{p_k} \;\Leftrightarrow\; r = x_k,\; x_k = x_{p_k},\; x_{p_k} = r.$$

Поскольку каждая матрица E_k является симметричной и $P^T = E_1 E_2 \cdots E_n$, то также можно выполнить замещение вектора x на вектор $P^T x$:

$$\textbf{for } k = n : 1 : -1$$
$$x_k \leftrightarrow x_{p_k}$$

Существуют разные стратегии выбора ведущего элемента. Мы остановимся на стратегии частичного выбора. Пусть матрица

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}.$$

Чтобы добиться наименьших множителей в первой матрице разложения по Гауссу с помощью взаимных перестановок строк, надо сделать элемент a_{11} наибольшим в первом столбце. Если E_1 — матрица взаимных перестановок, тогда

$$E_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Поэтому

$$E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{bmatrix}$$

и

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix} \Rightarrow M_1 E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{bmatrix}.$$

Теперь, чтобы получить наименьший множитель в матрице M_2, необходимо переставить 2-ю и 3-ю строки и т.д.

Пример иллюстрирует общую идею, основанную на перестановке строк. Обобщая эту идею, получим следующий алгоритм:

LU-разложение с частичным выбором

Если матрица $E \in \mathbb{R}^{n\times n}$, то данный алгоритм вычисляет матрицы преобразования Гаусса $M_1, M_2 \ldots, M_{n-1}$ и матрицы взаимных перестановок $E_1, E_2, \ldots, E_{n-1}$, такие что матрица $M_{n-1}E_{n-1}\cdots M_1 E_1 A = U$ является верхней треугольной. При этом нет множителей, превосходящих 1 по абсолютной величине. Подматрица $[a_{ik}]_{i=1}^{k}$ замещается на матрицу $[u_{ik}]_{i=1}^{k}$, $k = 1, 2, \ldots, n$. Подматрица $[a_{ik}]_{i=k+1}^{n}$ замещается на матрицу $[m_{k;ik}]_{i=k+1}^{n}$, $k = 1, 2, \ldots, n-1$. Целочисленный вектор piv размера $n-1$ задает взаимные перестановки. В частности, матрица E_k переставляет строки k и piv_k, $k = 1, 2, \ldots, n-1$.

for $k = 1 : n$

1. Зададим μ, такое что $k \le \mu \le n$ и $|a_{\mu k}| = \max_{k\le i\le n} |a_{ik}|$
2. $a_{k,k:n} \leftrightarrow a_{\mu,k:n}$; $piv_k = \mu$

if $a_{kk} \neq 0$

$t =$ `gauss` $(A_{k:n,k})$; $A_{k+1:n,k} = t$

$A_{k:n,k+1:n} =$ `gauss_app` $(A_{k:n,k+1:n}, t)$

end if

end for

Чтобы решить линейную систему $Ax = b$ после вызова последнего алгоритма, мы должны

1. Вычислить вектор $y = M_{n-1}E_{n-1}\cdots M_1E_1b$. 2. Решить верхнюю треугольную систему $Ux = y$.

2.1.2. Методы решения систем с симметричными матрицами

Здесь мы опишем методы, использующие специфику при решении задачи $Ax = b$. В случае, когда A — симметричная невырожденная матрица, т.е. $A = A^T$ и $\det(A) \neq 0$, существует разложение вида

$$A = LDL^T, \tag{2.4}$$

где L — нижняя унитреугольная матрица, D — диагональная матрица. В связи с этим работа связанная с получением разложения :eq:$sles - ldl$, составляет половину от того, что требуется для исключения Гаусса. Когда разложение :eq:$sles-ldl$ получено, решение системы $Ax = b$ может быть найдено посредством решения систем $Ly = b$ (прямая подстановка), $Dz = y$ и $L^Tx = z$.

LDL^T-разложение. Разложение (2.4) может быть найдено при помощи исключения Гаусса, вычисляющего $A = LU$, с последующим определением D из уравнения $U = DL^T$. Тем не менее можно использовать интересный альтернативный алгоритм непосредственного вычисления L и D.

Допустим, что мы знаем первые $j - 1$ столбцов матрицы L, диагональные элементы $d_1, d_2, \ldots, d_{j-1}$ матрицы D для некоторого j, $1 \leq j \leq n$. Чтобы получить способ вычисления l_{ij}, $i = j + 1, j + 2, \ldots, n$, и d_j приравняем j-е столбцы в уравнении $A = LDL^T$. В частности,

$$A(1 : j, j) = Lv, \tag{2.5}$$

где

$$v = DL^T e_j = \begin{bmatrix} d_1 l_{j1} \\ \vdots \\ d_{j-1} l_{jj-1} \\ d_j \end{bmatrix}.$$

Следовательно, компоненты v_k, $k = 1, 2, \ldots, j-1$ вектора v могут быть получены простым масштабированием элементов j-й строки матрицы L. Формула для j-й компоненты вектора v получается из j-го уравнения системы $L(1:j, 1:j)v = A(1:j, j)$:

$$v_j = a_{jj} - \sum_{k=1}^{j-1} l_{ik} v_k,$$

Когда мы знаем v, мы вычисляем $d_j = v_j$. «Нижняя» половина формулы (2.5) дает уравнение

$$L(j+1:n, 1:j)v(1:j) = A(j+1:n, j),$$

откуда для вычисления j-го столбца матрицы L имеем:

$$L(j+1:n, j) = (A(j+1:n, j) - L(j+1:n, 1:j-1)v(1:j-1))/v_j.$$

Реализация. Для получения LDL^T-разложения матрицы A можем написать функцию (сценарий ld.py[1]):

```
def ld(A):
    """
    Для симметричной матрицы A вычисляет нижнюю треугольную
    матрицу L и диагональную матрицу D, такие
    что A = LDL^T. Элементы a_{ij} замещаются    на l_{ij}, если i > j,
    и на d_i, если i = j
    """

    n = len(A)
    LD = np.array(A, float)
    for j in range(n):
        v = np.zeros(j+1)
        v[:j] = LD[j,:j]*LD[range(j),range(j)]
        v[j] = LD[j,j] - np.dot(LD[j,:j],v[:j])
        LD[j,j] = v[j]
        LD[j+1:,j] = (LD[j+1:,j] - np.dot(LD[j+1:,:j],v[:j]))/v[j]

    return LD
```

[1] src-sles/ld.py

В этой реализации мы использовали векторизованные вычисления. Разберем некоторые выражения. Строка

```
v[:j] = LD[j,:j]*LD[range(j),range(j)]
```

можно заменить следующим циклом:

```
for i in range(j):
    v[i] = LD[j,i]*LD[i,i]
```

В нашей программе доступ к j диагональным элементам массива A осуществляется выражением A[range(j),range(j)].

При вычислении v[j] использовалась функция np.dot, которая вычисляет скалярное произведение векторов.

Отметим также строку

```
LD[j+1:,j] = (LD[j+1:,j] - np.dot(LD[j+1:,:j],v[:j]))/v[j]
```

в которой используется срез L[j+1:,j], т.е. элементы с j+1-го до последнего в j-ом столбце.

Для решения системы $Ax = b$‘ с использованием LDL^T-разложения можно написать следующую функцию

```
def ld_solve(A, b):
    """
    Решает систему Ax = b с использованием LDL^T-разложения
    """
    LD = ld(A)
    b = np.array(b,float)
    for i in range(1, len(b)):
        b[i] = b[i] - np.dot(LD[i,:i],b[:i])
    b[:] = b[:]/LD[range(len(b)),range(len(b))]
    for i in range(len(b)-1, -1, -1):
        b[i] = (b[i] - np.dot(LD[i+1:,i],b[i+1:]))
    return b
```

Разложение Холецкого. Известно, что в случае симметричной положительно определенной матрицы разложение (2.4) существует и устойчиво. Тем не менее в этом случае можно использовать другое разложение:

$$A = GG^T \tag{2.6}$$

известное как *разложение Холецкого*, а матрицы G называются *треугольниками Холецкого*.

Это легко показать, исходя из существования LDL^T разложения. Так как для симметричной положительно определенной матрицы существует $A = LDL^T$ и диагональные элементы матрицы D положительны, то $G = L\mathrm{diag}(\sqrt{d_{11}}, \sqrt{d_{22}}, \ldots, \sqrt{d_{nn}})$.

2.2. Итерационные методы решения систем линейных алгебраических уравнений

2.2.1. Стандартные итерационные методы

В разделах 2.1.1 и 2.1.2 процедуры решения систем алгебраических уравнений были связаны с разложением матрицы коэффициентов A. Методы такого типа называются *прямыми методами*. Противоположностью прямым методам являются *итерационные методы*. Эти методы порождают последовательность приближенных решений $\{x^{(k)}\}$. При оценивании качества итерационных методов в центре внимания вопрос от том, как быстро сходятся итерации $x^{(k)}$.

Итерации Якоби и Гаусса — Зейделя. Простейшей итерационной схемой, возможно, являются *итерации Якоби*. Они определяются для матриц с ненулевыми диагональными элементами. Идею метода можно представить, используя запись 3×3-системы $Ax = b$ в следующем виде:

$$
\begin{aligned}
x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}, \\
x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}, \\
x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}.
\end{aligned}
$$

Предположим, что $x^{(k)}$ — какое-то приближение к $x = A^{-1}b$. Чтобы получить новое приближение $x^{(k+1)}$, естественно взять:

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11},$$
$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22},$$
$$x_3^{(k+1)} = (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33}.$$

Эти формулы и определяют итерации Якоби в случае $n = 3$. Для произвольных n мы имеем

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) /a_{ii}, \quad i = 1, 2, \ldots, n. \quad (2.7)$$

Заметим, что в итерациях Якоби при вычислении $x_i^{(k+1)}$ не используется информация, полученная в самый последний момент. Например, при вычислении $x_2^{(k+1)}$ используется $x_1^{(k)}$, хотя уже известна компонента $x_1^{(k+1)}$. Если мы пересмотрим итерации Якоби с тем, чтобы всегда использовать самые последние оценки для x_i, то получим:

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) /a_{ii}, \quad i = 1, 2, \ldots, n. \quad (2.8)$$

Так определяется то, что называется *итерациями Гаусса — Зейделя*.

Для итераций Якоби и Гаусса — Зейделя переход от $x^{(k)}$ к $x^{(k+1)}$ в сжатой форме описывается в терминах матриц L, D и U, определяемых следующим образом:

$$L = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & \cdots & \cdots & 0 \\ a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn-1} & 0 \end{bmatrix},$$

$$D = \operatorname{diag}(a_{11}, a_{12}, \ldots, a_{nn}),$$

$$U = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n-1n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Шаг Якоби имеет вид $M_J x^{(k+1)} = N_J x^{(k)} + b$, где $M_J = D$ и $N_J = -(L + U)$. С другой стороны, шаг Гаусса — Зейделя определяется как $M_G x^{(k+1)} = N_G x^{(k)} + b$, где $M_G = (D + L)$ и $N_G = -U$.

Процедуры Якоби и Гаусса — Зейделя — это типичные представители большого семейства итерационных методов, имеющих вид

$$Mx^{(k+1)} = Nx^{(k)} + b, \tag{2.9}$$

где $A = M - N$ — расщепление матрицы A. Для практического применения итераций (2.9) должна «легко» решаться система с матрицей M. Заметим, что для итераций Якоби и Гаусса — Зейделя матрица M соответственно диагональная и нижняя треугольная.

Сходятся ли итерации (2.9) к $x = A^{-1}b$, зависит от собственных значений матрицы $M^{-1}N$. Определим *спектральный радиус* произвольной $n \times n$-матрицы G как

$$\rho(G) = \max\{|\lambda| : \lambda \in \lambda(G)\},$$

тогда если матрица M невырожденная и $\rho(M^{-1}N) < 1$, то итерации $x^{(k)}$, определенные согласно $M^{(k+1)} = Nx^{(k)} + b$, сходятся к $x = A^{-1}b$ при любом начальном векторе $x^{(0)}$.

Последовательная верхняя релаксация. Метод Гаусса — Зейделя очень привлекателен в силу своей простоты. К несчастью, если спектральный радиус для $M_G^{-1}N_G$ близок к единице, то метод может оказаться непозволитель-

но медленным из-за того, что ошибки стремятся к нулю как $\rho(M_G^{-1}N_G)^k$. Чтобы исправить это, возьмем $\omega \in \mathbb{R}$ и рассмотрим следующую модификацию шага Гаусса — Зейделя:

$$x_i^{(k+1)} = \omega \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii} + (1-\omega) x_i^{(k)}. \tag{2.10}$$

Так определяется метод *последовательной верхней релаксации* (SOR — Successive Over Relaxation). В матричных обозначениях шаг SOR выглядит как

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b,$$

где $M_\omega = D + \omega L$ и $N_\omega = (1-\omega)D - \omega U$. Для небольшого числа специфических задач значения реалксационного параметра ω, минимизирующего $\rho(M_\omega^{-1}N_\omega)$, является известным. В более сложных задачах, однако, для того чтобы определить подходящее ω, может возникнуть необходимость в выполнении весьма трудного анализа собственных значений.

2.2.2. Метод сопряженных градиентов

Трудность, связанная с SOR и такого же типа методами, заключается в том, что они зависят от параметров, правильный выбор которых иногда бывает затруднителен. Например, для того чтобы чебышевское ускорение было успешным, нам нужны хорошие оценки для наибольшего и наименьшего собственных значений соответствующей итерационной матрицы ^{-1}N. Если эта матрица не устроена по-особому, то получение их в аналитическом виде, скорее всего, невозможно, а вычисление дорого.

Наискорейший спуск. Вывод метода связан с минимизацией функционала:

$$\varphi(x) = \frac{1}{2} x^T A x - x^T b,$$

где $b \in \mathbb{R}^n$ и матрица A предполагается положительно определенной и симметричной. Минимальное значение φ равно $-b^T A^{-1} b/2$ и достигается при $x = A^{-1}b$. Таким образом, минимизация φ и решение системы $Ax = b$ — эквивалентные задачи.

Одной из самых простых стратегий минимизации функционала φ является *метод наискорейшего спуска*. В текущей точке x_c функция φ убывает наиболее быстро в направлении антиградиента $\nabla\varphi(x_c) = b - Ax_c$. Мы называем $r_c = b - Ax_c$ *невязкой* вектора x_c. Если невязка ненулевая, то $\varphi(x_c + \alpha r_c) < \varphi(x_c)$ для некоторого положительного α (будем называть этот параметр *поправкой*). В методе наискорейшего спуска (с точной минимизацией на прямой) мы берем поправку

$$\alpha = \frac{r_c^T r_c}{r_c^T A r_c},$$

дающую минимум для $\varphi(x_c + \alpha r_c)$. Итерационный процесс запишется следующим образом

$$\begin{aligned} \alpha_k &= \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}}, \\ x_k &= x_{k-1} + \alpha_k r_{k-1}, \\ r_k &= b - Ax_k, \quad k = 1, 2, \ldots \end{aligned}$$

при начальных векторах $x_0 = 0$, $r_0 = b$.

К несчастью, скорость сходимости может быть недопустимо медленной, если число обусловленности $\kappa(A) = \lambda_1(A)/\lambda_2(A)$ большое. В этом случае линии уровня для φ являются сильно вытянутыми гиперэллипсоидами, а минимизация соответствует поиску самой нижней точки на относительно плоском дне крутого оврага. При наискорейшем спуске мы вынуждены переходить с одной стороны оврага на другую вместо того, чтобы спуститься к его дну. Направления градиента, возникающие при итерациях, являются слишком близкими; это и замедляет продвижение к точке минимума.

Произвольные направления спуска. Чтобы избежать ловушек при наискорейшем спуске, мы рассмотрим последовательную минимизацию φ вдоль какого-либо множе-

ства направлений $\{p_1, p_2, \ldots\}$, которые не обязаны соответствовать невязкам $\{r_0, r_1, \ldots\}$. Легко показать, что минимум $\varphi(x_{k-1} + \alpha p_k)$ по α дает

$$\alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}.$$

Для того, чтобы обеспечить уменьшение функционала φ, мы должны потребовать, чтобы p_k не был ортогонален к r_{k-1}. Проблема состоит в том, как выбирать эти векторы, чтобы гарантировать глобальную сходимость и в то же время обойти ловушки наискорейшего спуска.

Метод сопряженных градиентов. Как было сказано выше направления спуска p_k нужно выбирать так, чтобы они не были ортогональны к невязкам r_{k-1}, т.е. $p_k r_{k-1} \neq 0$. Кроме того, метод сопряженных градиентов основан на том, что требуется, чтобы направление p_k было A-сопряженным по отношению к $p_1, p_2, \ldots, p_{k-1}$, т.е. $p_m^T A p_k = 0$ для $m = 1, 2, \ldots, k-1$.

Поскольку наша цель — осуществить быстрое сокращение величины невязок, естественно выбирать в качестве p_k вектор, который ближе всего к r_{k-1} среди векторов, A-сопряженных с $p_1, p_2, \ldots, p_{k-1}$.

Для получения таких направлений спуска и нахождения приближенного решения используется *метод сопряженных градиентов*. Ниже представлен код функции, реализующий данный алгоритм (файл cg.py[2])

```
def cg(A, b, tol, it_max):
    it = 0
    x = 0
    r = np.copy(b)
    r_prev = np.copy(b)
    rho = np.dot(r, r)
    p = np.copy(r)
    while (np.sqrt(rho) > tol*np.sqrt(np.dot(b, b)) and it < it_max):
        it += 1
        if it == 1:
            p[:] = r[:]
        else:
            beta = np.dot(r, r)/np.dot(r_prev, r_prev)
            p = r + beta*p
```

[2] src-sles/cg.py

```
            w = np.dot(A, p)
            alpha = np.dot(r, r)/np.dot(p, w)
            x = x + alpha*p
            r_prev[:] = r[:]
            r = r - alpha*w
            rho = np.dot(r, r)

    return x, it
```

2.3. Тестирование реализации методов

2.4. Задачи

Задача 1: Решение системы линейных уравнений с трехдиагональной матрицей

Написать программу, которая решает систему линейных уравнений для трехдиагональной ($a_{ij} = 0$ при $|i-j| > 1$) $n \times n$-матрицы на основе LU-разложения. Написать следующие тестовые функции:

1. Найти решение уравнения с

$$a_{ii} = 2, \quad a_{ii-1} = a_{ii+1} = -1$$

при правой части $b_i = 2h^2$, $h = 1/n$, $i = 1, 2, \ldots, n-1$, $b_n = -(n-1) * h(1-(n-1)/h)$ и сравнить его с точным решением $x_i = ih(1-ih)$, $i = 1, 2, \ldots, n$.

2. Вычислить определитель матрицы и сравнить его значение с точным $n+1$.

Подсказка. Трехдиагональная матрица A задается тремя диагоналями:

$$d_i = a_{ii}, \quad e_i^u = a_{ii+1}, \quad e_i^l = a_{ii-1}.$$

В модуле функция (например, `lu3`) выполняет LU-разложение матрицы A и возвращает результат в виде трех диагоналей. Для решения системы используется другая функция (например, `solve_lu3`).

Задача 2: Метод Гаусса с частичным выбором ведущего элемента

Написать модуль, который реализует идеи частичного выбора ведущего элемента из раздела 2.1.1. Функция для LU-разложения должна выводить, кроме самого разложения, еще и вектор, определяющий матрицу перестановок. Напишите тестовые функции для проверки выполнения LU-разложения и решения системы уравнений с матрицей

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}$$

Задача 3: Разложение Холецкого

Написать программу, реализующую разложение Холецкого $A = GG^T$ для симметричной положительно определенной матрицы A и вычисляющей определитель матрицы на основе этого разложения. Найти разложение Холецкого и определитель матрицы Гильберта, для которой

$$a_{ij} = \frac{1}{i+j-1}, \quad i = 1, 2, \ldots, n, \ j = 1, 2, \ldots, n$$

при $n = 4$.

Задача 4: Метод Якоби

Написать программу, реализующую метод Якоби с использованием циклов Python (функция `jacobi`) и с векторизованными вычислениями (функция `jacobi_vec`). Сравнить время выполнения этих функций. Написать тестовые функции, проверяющие работу функции `jacobi`.

Задача 5: Метод Зейделя

Написать программу, реализующую метод Зейделя (функция `seidel`). Написать тестовые функции, проверяющие работу функции `seidel`.

Задача 6: Сравнение методов Якоби и Зейделя

Используя функции из 4 и 5, найти решение задачи системы $Ax = b$ с трехдиагональной матрицей A, в которой

$$a_{ii} = 2, \quad a_{ii+1} = -1 - \alpha, \quad a_{ii-1} = -1 + \alpha, \quad i = 1, 2 \ldots, n-1,$$

$$a_{00} = 2, \quad a_{01} = -1 - \alpha, \quad a_{n-1n} = -1 + \alpha, \quad a_{nn} = 2,$$

а правая часть

$$b_0 = 1 - \alpha, \quad b_i = 0, \quad i = 1, 2, \ldots, n-1, \quad b_n = 1 + \alpha,$$

определяет точное решение $x_i = 1$, $i = 1, 2, \ldots, n$. Сравнить скорости сходимости (число итераций) методов Якоби и Зейделя при различных параметрах n и α при $0 \leqslant \alpha \leqslant 1$. Для этого построить график зависимости числа итераций K от n при фиксированном α, а также график зависимости числа итераций K от α при фиксированном n.

Задача 7: Метод верхней релаксации

Написать программу, реализующую приближенное решение системы линейных алгебраических уравнений методов релаксации из 2.2.1. Написать тестовые функции. Исследовать графически зависимость скорости сходимости этого итерационного метода от итерационного параметра ω при численном решении системы уравнений из 6 при различных параметрах n и α.

Задача 8: Метод сопряженных градиентов

С помощью метода сопряженных градиентов (файл cg.py[3]) найти решение системы $Ax = b$ с матрицей Гильберта из задачи 5 и правой частью

$$b_i = \sum_{j=1}^{n} a_{ij}, \quad i = 1, 2, \ldots, n,$$

[3] `src-sles/cg.py`

для которой точное решение есть $x_i = 1$, $i = 1, 2, \ldots, n$. Построить график зависимости числа итераций от n.

Предметный указатель