



$$C = A \times B$$
$$c_{ij} = \sum_{r=1}^n a_{ir} \times b_{rj}$$

$$A_{ij} = (-1)^{i+j} M_{ij}$$
$$A^{*T} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$
$$A^{-1} = \frac{1}{\det(A)} \times A^{*T}$$

Линейная Алгебра на Python

$$A_{rot-cl} = \begin{pmatrix} \cos \frac{\pi}{4} & -\sin \frac{\pi}{4} \\ -\sin \frac{\pi}{4} & \cos \frac{\pi}{4} \end{pmatrix}$$

$$f(\lambda) = |A - \lambda E|$$

Devpractice Team

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$
$$A^{*T} = \begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \dots & \dots & \dots & \dots \\ A_{1n} & A_{2n} & \dots & A_{nn} \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2.2 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & -1 & 0 \\ 1.5 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$
$$A - \lambda E = \begin{pmatrix} a_{11} - \lambda & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} - \lambda & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} - \lambda \end{pmatrix}$$

$$f(z) = A_{m1} \times z = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 2 \end{pmatrix}^T$$

$$A^{-1} = \frac{1}{\det(A)} \times A^{*T}$$

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a'_{22}x_2 + \dots + a'_{2n}x_n = b_2 \\ \dots \\ a''_{nn}x_n = b_n \end{cases}$$
$$|A - \lambda E| = \begin{vmatrix} 7 - \lambda & 3 \\ -5 & -1 - \lambda \end{vmatrix} = (7 - \lambda) \cdot (-1 - \lambda) - (-15) = \lambda^2 - 6\lambda + 8$$

$$A_{rot-cl} = \begin{pmatrix} \cos \frac{\pi}{4} & -\sin \frac{\pi}{4} \\ -\sin \frac{\pi}{4} & \cos \frac{\pi}{4} \end{pmatrix}$$

УДК 512.64+004.4

ББК: 22.143+32.973

Devpractice Team. Линейная алгебра на Python. - devpractice.ru. 2019. - 114 с.: ил.

Книга “Линейная алгебра на Python” - это попытка соединить две области: математику и программирование. В ней вы познакомитесь с базовыми разделами линейной алгебры и прекрасным инструментом для решения задач - языком программирования Python. Основные разделы книги посвящены матрицам и их свойствам, решению систем линейных уравнений, векторам, разложению матриц и комплексным числам.

УДК 512.64+004.6

ББК: 22.143+32.973

Материал составили и подготовили:

Абдрахманов М.И.

Мамонов И.А.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, представленный в книге, многократно проверен. Но поскольку человеческие и технические ошибки все же возможны, авторы и коллектив проекта *devpractice.ru* не несут ответственности за возможные ошибки и последствия, связанные с использованием материалов из данной книги.

© devpractice.ru, 2019

© Абдрахманов М.И., 2019

Оглавление

Введение.....	4
О чем эта книга.....	4
Установка необходимого набора программного обеспечения.....	4
Установка <i>Python</i>	4
Установка <i>Python</i> в <i>Windows</i>	5
Установка <i>Python</i> в <i>Linux</i>	6
Установка <i>Anaconda</i>	6
Установка <i>Anaconda</i> в <i>Windows</i>	6
Установка <i>Anaconda</i> в <i>Linux</i>	7
Установка библиотек <i>Numpy</i> и <i>Scipy</i>	8
Проверка работоспособности.....	8
Уроки по языку программирования <i>Python</i>	9
Глава 1. Матрицы.....	10
1.1 Общие понятия.....	10
1.2 Виды матриц и способы их создания в <i>Python</i>	10
1.2.1 Вектор.....	11
1.2.1.1 Вектор-строка.....	11
1.2.1.2 Вектор-столбец.....	12
1.2.3 Диагональная матрица.....	14
1.2.4 Единичная матрица.....	15
1.2.5 Нулевая матрица.....	16
1.2.6 Задание матрицы в общем виде.....	17
1.3 Транспонирование матрицы.....	17
1.4 Действия над матрицами.....	21
1.4.1 Умножение на число.....	21
1.4.2 Сложение.....	25
1.4.3 Умножение матриц.....	27
1.5 Определитель матрицы.....	32
1.6 Обратная матрица.....	39
1.7 Ранг матрицы.....	42
Глава 2. Решение систем линейных уравнений.....	44
2.1 Решение неоднородной системы в матричной форме.....	44
2.2 Метод Крамера.....	47
2.3 Метод Гаусса.....	51
2.4 Решение системы линейных уравнений на <i>Python</i>	53
Глава 3. Основы векторной алгебры.....	54
3.1 Векторные и скалярные величины.....	54
3.2 Проекция вектора на ось.....	55
3.3 Координаты вектора.....	55
3.4 Операции над векторами.....	58
3.4.1 Сложение векторов.....	58
3.4.2 Вычитание векторов.....	60
3.4.4 Скалярное произведение двух векторов.....	62
3.4.5 Векторное произведение двух векторов.....	63
3.4.6 Смешанное произведение трех векторов.....	65
Глава 4. Линейные векторные пространства.....	67
4.1 Общие сведения.....	67
4.2 Линейная зависимость векторов.....	71

4.3 Базис системы векторов.....	73
4.4 Линейные операторы.....	76
4.4.1 Отражение вектора в себя.....	78
4.4.2 Масштабирование.....	79
4.4.3 Отражение.....	81
4.4.3.1 Горизонтальное отражение.....	81
4.4.3.2 Вертикальное отражение.....	82
4.4.4 Поворот.....	83
4.4.4.1 По часовой стрелке.....	83
4.4.4.2 Против часовой стрелке.....	84
4.4.5 Перемещение по осям.....	85
4.4.6 Эквивалентные преобразования применительно к СЛАУ.....	87
4.5 Собственные векторы линейного оператора.....	91
4.5.1 Характеристический полином матрицы.....	91
4.5.2 Собственные векторы и собственные значения линейного оператора.....	93
Глава 5. Разложения матриц.....	96
5.1 <i>LU</i> -разложение матрицы.....	96
5.2 <i>QR</i> -разложение матрицы.....	99
5.3 Сингулярное разложение матрицы.....	102
Глава 6. Комплексные числа.....	104
6.1 Что такое комплексное число?.....	104
6.2 Задание комплексных чисел.....	106
6.2.1 Алгебраическая форма задания комплексного числа.....	106
6.2.2 Геометрическая интерпретация комплексного числа.....	106
6.2.3 Модуль и аргумент комплексного числа.....	107
6.2.4 Тригонометрическая форма задания комплексного числа.....	107
6.2.5 Показательная форма комплексного числа.....	108
6.3 Задание комплексного числа в <i>Python</i>	108
6.4 Комплексно сопряженное число.....	109
6.5 Операции над комплексными числами.....	110
6.5.1 Сложение комплексных чисел.....	110
6.5.2 Вычитание комплексных чисел.....	111
6.5.3 Умножение комплексных чисел.....	111
6.5.4 Деление комплексных чисел.....	112
6.6 Возведение в степень комплексного числа.....	112
6.7 Извлечение корня из комплексного числа.....	113
Список литературы.....	114
Алгебра / Линейная алгебра.....	114
Python.....	114

Введение

О чем эта книга

Предлагаем вашему вниманию книгу, в которой мы постарались соединить две области: программирование и математику. Изначально она задумывалась, как пособие для студентов технических вузов, которое бы позволило не только повторить важные темы из линейной алгебры, но и познакомиться с прекрасным инструментом для решения задач - языком программирования *Python* и необходимым набором библиотек. Но в процессе работы стало ясно, что в рамках программы обычного технического вуза (не федерального уровня), линейная алгебра, чаще всего присутствует как составная часть курса высшей математики, в рамках которой проходят только матрицы, векторную алгебру и решение систем линейных уравнений. Другие темы линейной алгебры, как правило, остаются без внимания. Было принято решение расширить список тем, но идейно книгу оставить той же. Концепция подачи материала выглядит так: вначале излагается теория, потом дается численный пример и решение задачи на языке программирования *Python*.

Мы будем очень рады, если книга окажется для вас полезной и интересной. А сейчас приступим к установке программного обеспечения, которое нам понадобится в работе.

Установка необходимого набора программного обеспечения

Для выполнения примеров из книги вам понадобится интерпретатор языка программирования *Python* и библиотеки *Numpy* и *Scipy*, которые содержат функции и структуры подходящие для решения задач линейной алгебры.

Вы можете установить *Python*, а потом воспользоваться специальной командой *pip* для развертывания *Numpy* и *Scipy*, либо установить пакет *Anaconda*, который уже содержит все нужные библиотеки.

Установка *Python*

На сегодняшний день существуют две версии *Python* – это *Python 2* и *Python 3*, у них отсутствует полная совместимость друг с другом. В нашей с вами работе, мы будем использовать *Python 3*.

Для установки интерпретатора *Python* на ваш компьютер, первое, что нужно сделать – это скачать дистрибутив. Загрузить его можно с официального сайта, перейдя по ссылке <https://www.python.org/downloads/>.

Установка *Python* в *Windows*

Для операционной системы *Windows* дистрибутив *Python* распространяется либо в виде исполняемого файла (с расширением *exe*), либо в виде архивного файла (с расширением *zip*). Если вы используете *Windows 7*, не забудьте установить *Service Pack 1*!



Рисунок В1 — Страница сайта <https://www.python.org/>

Скачайте дистрибутив, запустите его и пройдите все этапы процесса установки.

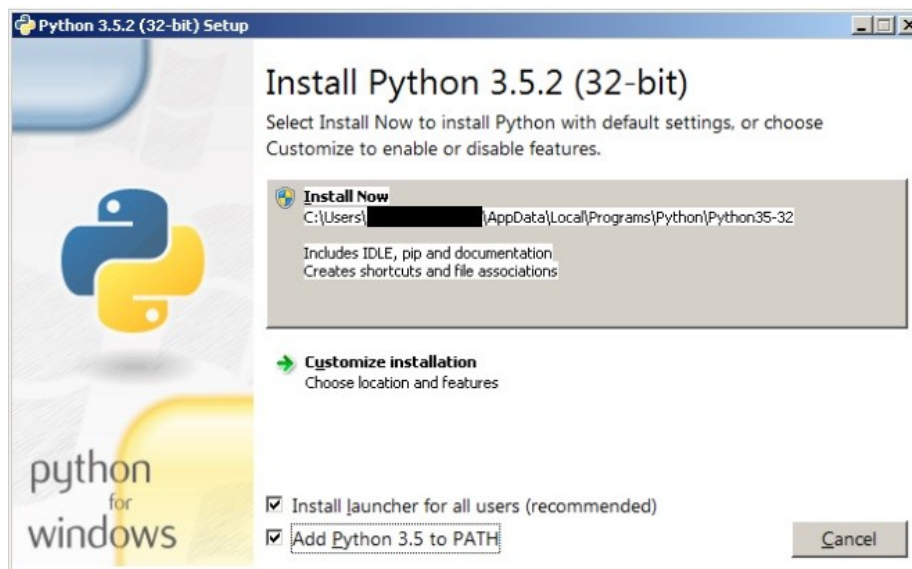


Рисунок В2 — Установка интерпретатора *Python*

Установка *Python* в *Linux*

Чаще всего интерпретатор *Python* уже входит в состав *Linux*. Это можно проверить набрав в терминале:

```
> python
```

или команду:

```
> python3
```

В первом случае, вы запустите *Python* 2 во втором – *Python* 3. В будущем во всех дистрибутивах *Linux*, включающих *Python*, будет входить только третья версия. Если у вас, при попытке запустить *Python*, выдается сообщение о том, что он не установлен, или установлен, но не тот, что вы хотите, то у вас есть два пути:

- а) собрать *Python* из исходников;
- б) взять из репозитория.

Для установки *Python* из репозитория в *Ubuntu*, воспользуйтесь командой:

```
> sudo apt-get install python3
```

Более подробную информацию по установке можете найти на странице <https://www.python.org/downloads/>.

Установка *Anaconda*

Для удобства запуска примеров и изучения языка *Python*, советуем установить на свой компьютер пакет *Anaconda*. Этот пакет включает в себя интерпретатор языка *Python* (есть версии 2 и 3), большой набор библиотек для научных расчетов и удобную среду разработки и исполнения, запускаемую в браузере. Для установки *Anaconda*, скачайте дистрибутив со страницы <https://www.continuum.io/downloads>, там вы можете найти пакеты под *Windows*, *Linux* и *MacOS*.

Установка *Anaconda* в *Windows*

Запустите скачанный дистрибутив *Anaconda* и выполните то, что требует от вас программа установки.



Рисунок В3 — Установка *Anaconda* в *Windows*

Установка *Anaconda* в *Linux*

Скачайте дистрибутив *Anaconda* для *Linux*, он будет иметь расширение *.sh*, и запустите установку командой:

```
> bash имя_дистрибутива.sh
```

В результате вы увидите приглашение к установке. Для продолжения процесса нажмите “Enter”.

```

Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
WARNING: The "syslog" option is deprecated
Enter tester's password:
session setup failed: NT_STATUS_LOGON_FAILURE
tester@tester-VirtualBox:/etc/samba$ cd ~
tester@tester-VirtualBox:~$ ls
code  dist  Downloads  Music  Public  Templates  vm
Desktop  Documents  lab  Pictures  temp  Videos  winetest
tester@tester-VirtualBox:~$ cd Do
bash: cd: Do: No such file or directory
tester@tester-VirtualBox:~$ cd Downloads/
tester@tester-VirtualBox:~/Downloads$ ls
Anaconda3-4.2.0-Linux-x86_64.sh  vim-colors-solarized-master.zip
codeschool-vim-theme-master    vim-distinguished-develop
codeschool-vim-theme-master.zip vim-distinguished-develop.zip
molokai.vim                    Vistaluna_Basic_by_Pgase.rar
vim-colors-solarized-master
tester@tester-VirtualBox:~/Downloads$ bash Anaconda3-4.2.0-Linux-x86_64.sh

Welcome to Anaconda3 4.2.0 (by Continuum Analytics, Inc.)

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>

```

Рисунок В4 — Установка *Anaconda* в *Linux*

Далее, необходимо будет принять лицензионное соглашение (его нужно пролистать до конца) и выбрать место установки.

Установка библиотек *Numpy* и *Scipy*

Если вы установили *Anaconda*, то в ее комплекте уже присутствуют нужные библиотеки. Если нет, то выполните в консоли команду:

```
> pip install numpy  
> pip install scipy
```

В *Linux*, для установки *Numpy* и *Scipy* для *Python 3*, может понадобится выполнить команды:

```
> pip3 install numpy  
> pip3 install scipy
```

Проверка работоспособности

Теперь проверим работоспособность всего того, что мы установили. Для начала протестируем интерпретатор в командном режиме. Если вы работаете в *Windows*, то найдите в меню *Пуск* среду *IDLE* и запустите её, появится соответствующая оболочка.

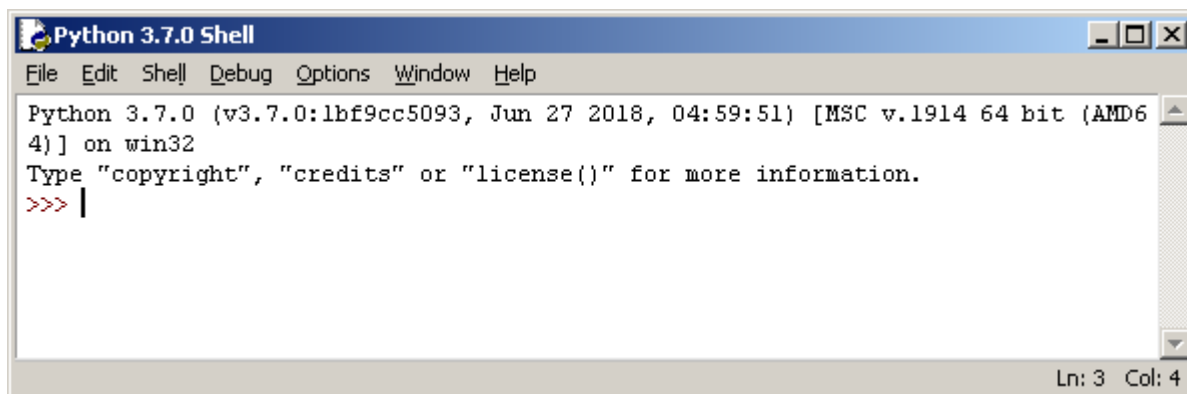
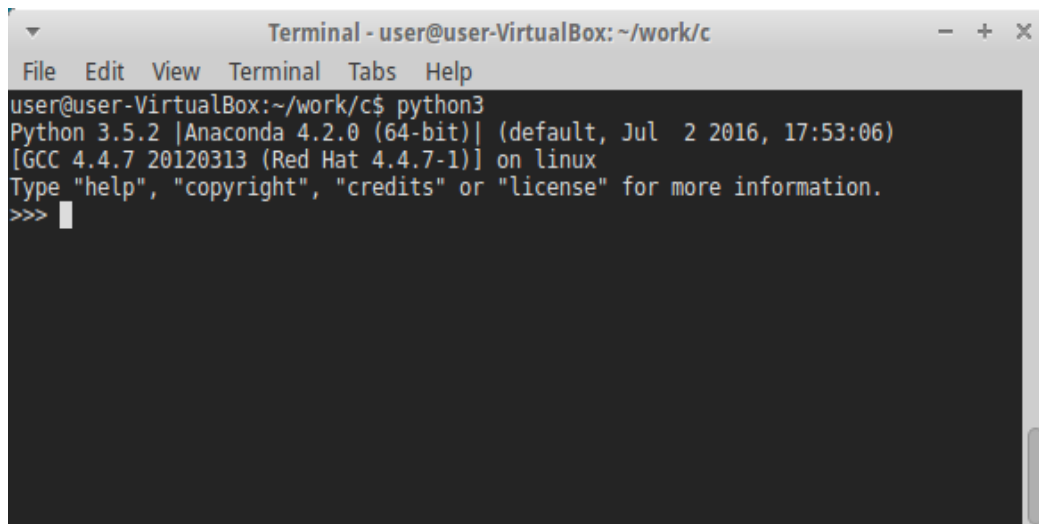


Рисунок В5 — Оболочка для работы с *Python* в командном режиме в *Windows*

Другой возможный вариант - это нажать сочетание *Win+R*, в появившемся окне ввести *python*.

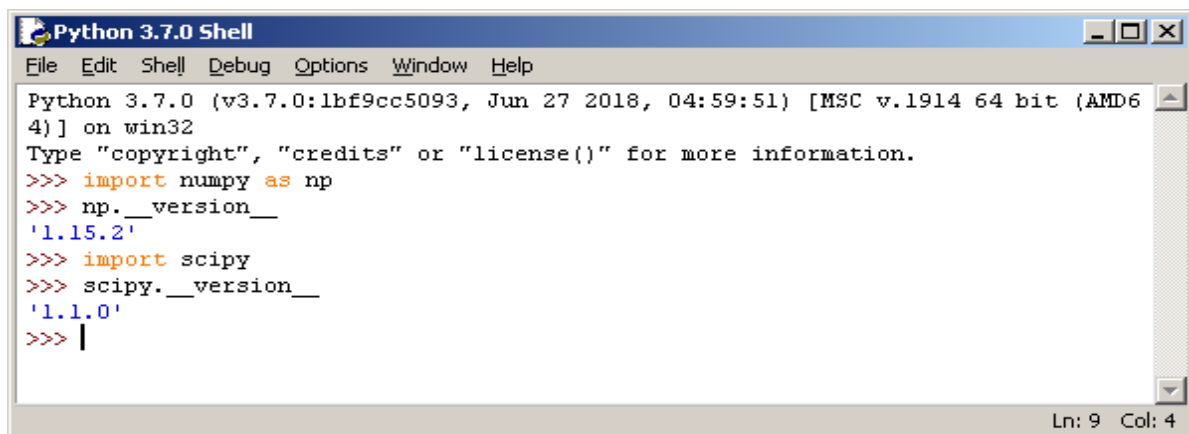
В *Linux* откройте окно терминала и введите в нем *python3* (или *python*). В результате *Python* запустится в командном режиме, выглядеть это будет примерно так как показано на рисунке В6.



```
Terminal - user@user-VirtualBox: ~/work/c
File Edit View Terminal Tabs Help
user@user-VirtualBox:~/work/c$ python3
Python 3.5.2 [Anaconda 4.2.0 (64-bit)] (default, Jul 2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

Рисунок В6 — Интерпретатор *Python*, запущенный в командном режиме, в *Linux*

В окне интерпретатора или *IDLE* введите команды для проверки версий библиотек *Numpy* и *Scipy*. Результат должен быть следующий:



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import numpy as np
>>> np.__version__
'1.15.2'
>>> import scipy
>>> scipy.__version__
'1.1.0'
>>> |
```

Рисунок В7 — Проверка версий библиотек *Numpy* и *Scipy*

Уроки по языку программирования Python

Если вы не знакомы с языком программирования Python, то мы рекомендуем пройти несколько первых уроков из списка <https://devpractice.ru/python-lessons/> или скачать книгу "Python. Уроки" по адресу <https://devpractice.ru/book-python-lessons/>. Книга и уроки распространяются абсолютно бесплатно!

Глава 1. Матрицы

1.1 Общие понятия

Матрицей называют объект, записываемый в виде прямоугольной таблицы, элементами которой являются числа (могут быть как действительные, так и комплексные). Пример матрицы:

$$M = \begin{pmatrix} 1 & 3 & 5 \\ 7 & 2 & 4 \end{pmatrix}.$$

В общем виде матрица записывается так:

$$M = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

Представленная выше матрица состоит из m -строк и n -столбцов. Каждый ее элемент имеет соответствующее позиционное обозначение, определяемое номером строки и столбца на пересечении которых он расположен: a_{ij} - находится в i -ой строке и j -м столбце.

Важным элементом матрицы является **главная диагональ**, ее составляют элементы, у которых совпадают номера строк и столбцов.

1.2 Виды матриц и способы их создания в *Python*

Так как матрица - это прямоугольная таблица чисел, то задание матрицы в *Python* будет предполагать построение соответствующего двумерного массива. Для создания такого массива из стандартных средств *Python* наиболее подходящим является тип данных **список** (англ. *list*). Но списки не самый удобный инструмент для выполнения операций с матрицами, для этих целей мы будем использовать тип *array* из библиотеки *Numpy*.

Для того, чтобы использовать библиотеку *Numpy* ее нужно предварительно установить, после этого - импортировать в свой проект. По установке *Numpy* можно подробно прочитать в разделе “Установка библиотек *Numpy* и *Scipy*” из введения.

Для того импорта данного модуля, добавьте в самое начало вашей программы следующую строку:

```
>>> import numpy as np
```

Если после импорта не было сообщений об ошибке, то значит все прошло удачно и можно начинать работу. *Numpy* содержит большое количество функций для работы с матрицами, которые мы будем активно использовать. Обязательно убедитесь в том, что библиотека установлена и импортируется в проект без ошибок.

Рассмотрим, различные виды матриц и способы их задания в *Python*.

1.2.1 Вектор

Вектором называется матрица, у которой есть только один столбец или одна строка. Более подробно свойства векторов, их геометрическая интерпретация и операции над ними будут рассмотрены в “Главе 3. Основы векторной алгебры”.

1.2.1.1 Вектор-строка

Вектор-строка имеет следующую математическую запись:

$$v = (1 \ 2).$$

Такой вектор в *Python* можно задать следующим образом:

```
>>> v_hor_np = np.array([1, 2])
>>> print(v_hor_np)
[1 2]
```

Если необходимо создать **нулевой** или **единичный вектор** (вектор, у которого все элементы либо нули, либо единицы), то можно использовать специальные функции из библиотеки *Numpy*.

Создадим нулевую вектор-строку, состоящую из пяти элементов:

```
>>> v_hor_zeros_v1 = np.zeros((5,))
>>> print(v_hor_zeros_v1)
[0. 0. 0. 0. 0.]
```

В случае, если требуется построить вектор-строку так, чтобы он сам являлся элементом массива, это может понадобиться если вектор, в последующем, нужно

будет транспонировать (см. раздел “1.3 Транспонирование матрицы”), то данная задача решается так:

```
>>> v_hor_zeros_v2 = np.zeros((1, 5))
>>> print(v_hor_zeros_v2 )
[[0. 0. 0. 0. 0.] ]
```

Аналогично построим единичную вектор-строку:

```
>>> v_hor_one_v1 = np.ones((5,))
>>> print(v_hor_one_v1)
[1. 1. 1. 1. 1.]
```

```
>>> v_hor_one_v2 = np.ones((1, 5))
>>> print(v_hor_one_v2)
[[1. 1. 1. 1. 1.] ]
```

1.2.1.2 Вектор-столбец

Вектор-столбец имеет следующую математическую запись:

$$v = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

В общем виде вектор столбец можно задать следующим образом:

```
>>> v_vert_np = np.array([[1], [2]])
>>> print(v_vert_np)
[[1]
 [2]]
```

Рассмотрим способы создания нулевых и единичных векторов-столбцов. Построим **нулевой вектор-столбец**:

```
>>> v_vert_zeros = np.zeros((5, 1))
>>> print(v_vert_zeros)
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

Едини́чный вектор-столбец можно создать с помощью функции *ones()*:

```
>>> v_vert_ones = np.ones((5, 1))
>>> print(v_vert_ones)
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

1.2.2 Квадратная матрица

Довольно часто, на практике, приходится работать с **квадратными матрицами**. Квадратной называется матрица, у которой количество столбцов и строк совпадает. В общем виде они выглядят так:

$$M_{sqr} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

В примерах будем работать со следующей матрицей:

$$M_{sqr} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

В *Numpy* можно создать квадратную матрицу с помощью метода ***array()***:

```
>>> m_sqr_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> print(m_sqr_arr)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Как вы уже наверное заметили, аргументом функции ***np.array()*** является список *Python*, его можно построить отдельно и передать в функцию:

```
>>> m_sqr = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> m_sqr_arr = np.array(m_sqr)
>>> print(m_sqr_arr)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

В *Numpy* есть еще один способ создания матриц - это построение объекта типа *matrix* с помощью одноименного метода. Задать матрицу можно в виде списка:

```
>>> m_sqr_mx = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> print(m_sqr_mx)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Также доступен стиль *Matlab*, когда между элементами ставятся пробелы, а строки разделяются точкой с запятой, при этом такое описание должно быть передано в виде строки:

```
>>> m_sqr_mx = np.matrix('1 2 3; 4 5 6; 7 8 9')
>>> print(m_sqr_mx)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

1.2.3 Диагональная матрица

Особым видом квадратной матрицы является **диагональная** - это такая матрица, у которой все элементы, кроме тех, что расположены на главной диагонали равны нулю:

$$Mdiag = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}.$$

Диагональную матрицу можно построить вручную, задав только значения элементам на главной диагонали:

```
>>> m_diag = [[1, 0, 0], [0, 5, 0], [0, 0, 9]]
>>> m_diag_np = np.matrix(m_diag)
>>> print(m_diag_np)
[[1 0 0]
 [0 5 0]
 [0 0 9]]
```

Библиотека *Numpy* предоставляет инструменты, которые могут упростить построение такой матрицы.

Первый вариант подойдет в том случае, если у вас уже есть матрица, а вы из нее хотите сделать диагональную. Создадим матрицу размера 3x3:

```
>>> m_sqr_mx = np.matrix('1 2 3; 4 5 6; 7 8 9')
```

Извлечем ее главную диагональ:

```
>>> diag = np.diag(m_sqr_mx)
>>> print(diag)
[1 5 9]
```

Построим диагональную матрицу на базе полученной диагонали:

```
>>> m_diag_np = np.diag(np.diag(m_sqr_mx))
>>> print(m_diag_np)
[[1 0 0]
 [0 5 0]
 [0 0 9]]
```

Второй вариант подразумевает построение единичной матрицы, ей будет посвящен следующий параграф.

1.2.4 Единичная матрица

Единичной матрицей называют квадратную матрицу, у которой элементы главной диагонали равны единицы, а все остальные нулю:

$$E = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{pmatrix}.$$

Создадим единичную матрицу на базе списка, который передадим в качестве аргумента функции **matrix()**:

```
>>> m_e = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> m_e_np = np.matrix(m_e)
>>> print(m_e_np)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Такой способ не очень удобен, к счастью для нас, для построения такого типа матриц в библиотеке *Numpy* есть специальная функция — **eye()**:

```
>>> m_eye = np.eye(3)
>>> print(m_eye)
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

В качестве аргумента функции передается размерность матрицы, в нашем примере - это матрица 3x3. Тот же результат можно получить с помощью функции **identity()**:

```
>>> m_idnt = np.identity(3)
>>> print(m_idnt)
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

1.2.5 Нулевая матрица

У **нулевой матрицы** все элементы равны нулю:

$$Z = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{pmatrix}.$$

Пример того, как создать такую матрицу с использованием списков, мы приводить не будем, он делается по аналогии с предыдущим разделом. Что касается *Numpy*, то в составе этой библиотеки есть функция **zeros()**, которая создает нужную нам матрицу:

```
>>> m_zeros = np.zeros((3, 3))
>>> print(m_zeros)
[[0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]]
```

В качестве параметра функции **zeros()** передается размерность требуемой матрицы в виде кортежа из двух элементов, первый из которых - число строк, второй - столбцов. Если функции **zeros()** передать в качестве аргумента число, то будет построена нулевая вектор-строка, это мы делали в параграфе, посвященном векторам.

1.2.6 Задание матрицы в общем виде

Если у вас уже есть данные о содержимом матрицы, то создать ее можно, используя списки *Python* или функцию ***matrix()*** из библиотеки *Numpy*:

```
>>> m_mx = np.matrix('1 2 3; 4 5 6')
>>> print(m_mx)
[[1 2 3]
 [4 5 6]]
```

Если же вы хотите создать матрицу заданного размера с произвольным содержимым, чтобы потом ее заполнить, проще всего для того использовать функцию ***zeros()***, которая создаст матрицу заданного размера, заполненную нулями:

```
>>> m_var = np.zeros((2, 5))
>>> print(m_var)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

1.3 Транспонирование матрицы

Транспонирование матрицы - это процесс замены строк матрицы на ее столбцы, а столбцов, соответственно, на строки. Полученная в результате матрица называется транспонированной. Символ операции транспонирования - буква T .

➤ Численный пример

Для исходной матрицы:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Транспонированная будет выглядеть так:

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

➤ Пример на Python

Решим задачу транспонирования матрицы на *Python*. Создадим матрицу A :

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> print(A)
[[1 2 3]
 [4 5 6]]
```

Транспонируем матрицу с помощью метода ***transpose()***:

```
>>> A_t = A.transpose()
>>> print(A_t)
[[1 4]
 [2 5]
 [3 6]]
```

Существует сокращенный вариант получения транспонированной матрицы, он удобен для практического использования:

```
>>> print(A.T)
[[1 4]
 [2 5]
 [3 6]]
```

Рассмотрим свойства транспонированных матриц. Операции сложения и умножения матриц, а также вычисление определителя будут рассмотрены в последующих параграфах, при первом прочтении можете пропустить этот раздел.

Свойство 1. Дважды транспонированная матрица равна исходной матрице:

$$(A^T)^T = A.$$

➤ Численный пример

$$\left(\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T \right)^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> print(A)
[[1 2 3]
 [4 5 6]]
>>> R = (A.T).T
>>> print(R)
[[1 2 3]
 [4 5 6]]
```

Свойство 2. Транспонированная сумма матриц равна сумме транспонированных матриц:

$$(A + B)^T = A^T + B^T.$$

➤ Численный пример

$$\left(\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \\ 0 & 7 & 5 \end{pmatrix} \right)^T = \begin{pmatrix} 8 & 10 & 12 \\ 4 & 12 & 11 \end{pmatrix}^T = \begin{pmatrix} 8 & 4 \\ 10 & 12 \\ 12 & 11 \end{pmatrix},$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T + \begin{pmatrix} 7 & 8 & 9 \\ 0 & 7 & 5 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 0 \\ 8 & 7 \\ 9 & 5 \end{pmatrix} = \begin{pmatrix} 8 & 4 \\ 10 & 12 \\ 12 & 11 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> B = np.matrix('7 8 9; 0 7 5')
>>> L = (A + B).T
>>> R = A.T + B.T
>>> print(L)
[[ 8  4]
 [10 12]
 [12 11]]
>>> print(R)
[[ 8  4]
 [10 12]
 [12 11]]
```

Свойство 3. Транспонированное произведение матриц равно произведению транспонированных матриц, расставленных в обратном порядке:

$$(AB)^T = B^T A^T.$$

➤ Численный пример

$$\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right)^T = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}^T = \begin{pmatrix} 19 & 43 \\ 22 & 50 \end{pmatrix},$$

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}^T \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}^T = \begin{pmatrix} 5 & 7 \\ 6 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 19 & 43 \\ 22 & 50 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> L = (A.dot(B)).T
>>> R = (B.T).dot(A.T)
>>> print(L)
[[19 43]
 [22 50]]
>>> print(R)
[[19 43]
 [22 50]]
```

В данном примере, для умножения матриц, использовалась функция **dot()** из библиотеки *Numpy*.

Свойство 4. Транспонированное произведение матрицы на число равно произведению этого числа на транспонированную матрицу:

$$(\lambda A)^T = \lambda A^T.$$

➤ Численный пример

$$\left(3 \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}\right)^T = \begin{pmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{pmatrix}^T = \begin{pmatrix} 3 & 12 \\ 6 & 15 \\ 9 & 18 \end{pmatrix},$$

$$3 \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = 3 \cdot \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} 3 & 12 \\ 6 & 15 \\ 9 & 18 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> k = 3
>>> L = (k * A).T
>>> R = k * (A.T)
>>> print(L)
[[ 3 12]
 [ 6 15]
 [ 9 18]]
>>> print(R)
[[ 3 12]
 [ 6 15]
 [ 9 18]]
```

Свойство 5. Определители исходной и транспонированной матрицы совпадают:

$$|A| = |A^T|.$$

➤ Численный пример

$$\det \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right) = 4 - 6 = -2,$$

$$\det \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}^T \right) = \det \left(\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \right) = 4 - 6 = -2.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> A_det = np.linalg.det(A)
>>> A_T_det = np.linalg.det(A.T)
>>> print(format(A_det, '.9g'))
-2
>>> print(format(A_T_det, '.9g'))
-2
```

Ввиду особенностей *Python* при работе с числами с плавающей точкой, в данном примере при вычислении определителя рассматриваются только первые девять значащих цифр после запятой (за это отвечает параметр `'.9g'`).

1.4 Действия над матрицами

1.4.1 Умножение на число

При умножении матрицы на число, все элементы матрицы умножаются на это число:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix},$$

$$C = \lambda \cdot A,$$

$$C = \begin{pmatrix} \lambda \cdot a_{11} & \lambda \cdot a_{12} & \dots & \lambda \cdot a_{1n} \\ \lambda \cdot a_{21} & \lambda \cdot a_{22} & \dots & \lambda \cdot a_{2n} \\ \dots & \dots & \dots & \dots \\ \lambda \cdot a_{m1} & \lambda \cdot a_{m2} & \dots & \lambda \cdot a_{mn} \end{pmatrix}.$$

➤ Численный пример

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

$$C = 3 \cdot A,$$

$$C = \begin{pmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> C = 3 * A
>>> print(C)
[[ 3  6  9]
 [12 15 18]]
```

Рассмотрим свойства операции умножения матрицы на число.

Свойство 1. Произведение единицы и любой заданной матрицы равно заданной матрице:

$$1 \cdot A = A.$$

➤ Численный пример

$$1 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> L = 1 * A
>>> R = A
>>> print(L)
[[1 2]
 [3 4]]
>>> print(R)
[[1 2]
 [3 4]]
```

Свойство 2. Произведение нуля и любой матрицы равно нулевой матрице, размерность которой равна исходной матрицы:

$$0 \cdot A = Z.$$

➤ Численный пример

$$0 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> Z = np.matrix('0 0; 0 0')
>>> L = 0 * A
>>> R = Z
>>> print(L)
[[0 0]
 [0 0]]
>>> print(R)
[[0 0]
 [0 0]]
```

Свойство 3. Произведение матрицы на сумму чисел равно сумме произведений матрицы на каждое из этих чисел:

$$(\alpha + \beta) \cdot A = \alpha \cdot A + \beta \cdot A.$$

➤ Численный пример

$$(2 + 3) \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = 2 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 3 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} + \begin{pmatrix} 3 & 6 \\ 9 & 12 \end{pmatrix} = \begin{pmatrix} 5 & 10 \\ 15 & 20 \end{pmatrix},$$

$$5 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 5 & 10 \\ 15 & 20 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> p = 2
>>> q = 3
>>> L = (p + q) * A
>>> R = p * A + q * A
>>> print(L)
[[ 5 10]
 [15 20]]
```

```
>>> print(R)
[[ 5 10]
 [15 20]]
```

Свойство 4. Произведение матрицы на произведение двух чисел равно произведению второго числа и заданной матрицы, умноженному на первое число:

$$(\alpha \cdot \beta) \cdot A = \alpha \cdot (\beta \cdot A).$$

➤ Численный пример

$$(2 \cdot 3) \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = 2 \cdot \left(3 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right) = 2 \cdot \begin{pmatrix} 3 & 6 \\ 9 & 12 \end{pmatrix} = \begin{pmatrix} 6 & 12 \\ 18 & 24 \end{pmatrix},$$

$$(2 \cdot 3) \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = 6 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 6 & 12 \\ 18 & 24 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> p = 2
>>> q = 3
>>> L = (p * q) * A
>>> R = p * (q * A)
>>> print(L)
[[ 6 12]
 [18 24]]
>>> print(R)
[[ 6 12]
 [18 24]]
```

Свойство 5. Произведение суммы матриц на число равно сумме произведений этих матриц на заданное число:

$$\lambda \cdot (A + B) = \lambda \cdot A + \lambda \cdot B.$$

➤ Численный пример

$$3 \cdot \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right) = 3 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 3 \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 18 & 24 \\ 30 & 36 \end{pmatrix},$$

$$3 \cdot \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix} = \begin{pmatrix} 18 & 24 \\ 30 & 36 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> k = 3
>>> L = k * (A + B)
>>> R = k * A + k * B
>>> print(L)
[[18 24]
 [30 36]]
>>> print(R)
[[18 24]
 [30 36]]
```

1.4.2 Сложение

Складывать можно только матрицы одинаковой размерности — то есть матрицы, у которых совпадает количество столбцов и строк.

➤ Численный пример

$$A = \begin{pmatrix} 1 & 6 & 3 \\ 8 & 2 & 7 \end{pmatrix}, B = \begin{pmatrix} 8 & 1 & 5 \\ 6 & 9 & 12 \end{pmatrix},$$

$$C = A + B,$$

$$C = \begin{pmatrix} 1+8 & 6+1 & 3+5 \\ 8+6 & 2+9 & 7+12 \end{pmatrix} = \begin{pmatrix} 9 & 7 & 8 \\ 14 & 11 & 19 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 6 3; 8 2 7')
>>> B = np.matrix('8 1 5; 6 9 12')
>>> C = A + B
>>> print(C)
[[ 9  7  8]
 [14 11 19]]
```

Рассмотрим свойства сложения матриц.

Свойство 1. Коммутативность сложения. От перестановки матриц их сумма не изменяется:

$$A + B = B + A.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> L = A + B
>>> R = B + A
>>> print(L)
[[ 6  8]
 [10 12]]
>>> print(R)
[[ 6  8]
 [10 12]]
```

Свойство 2. Ассоциативность сложения. Результат сложения трех и более матриц не зависит от порядка, в котором эта операция будет выполняться:

$$A + (B + C) = (A + B) + C.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \left(\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} + \begin{pmatrix} 1 & 7 \\ 9 & 3 \end{pmatrix} \right) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 6 & 13 \\ 16 & 11 \end{pmatrix} = \begin{pmatrix} 7 & 15 \\ 19 & 15 \end{pmatrix},$$
$$\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right) + \begin{pmatrix} 1 & 7 \\ 9 & 3 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix} + \begin{pmatrix} 1 & 7 \\ 9 & 3 \end{pmatrix} = \begin{pmatrix} 7 & 15 \\ 19 & 15 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> C = np.matrix('1 7; 9 3')
>>> L = A + (B + C)
>>> R = (A + B) + C
>>> print(L)
[[ 7 15]
 [19 15]]
>>> print(R)
[[ 7 15]
 [19 15]]
```

Свойство 3. Для любой матрицы A существует противоположная ей $(-A)$, такая, что их сумма является нулевой матрицей Z :

$$A + (-A) = Z.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + (-1) \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} -1 & -2 \\ -3 & -4 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> Z = np.matrix('0 0; 0 0')
>>> L = A + (-1)*A
>>> print(L)
[[0 0]
 [0 0]]
>>> print(Z)
[[0 0]
 [0 0]]
```

1.4.3 Умножение матриц

Умножение матриц это уже более сложная операция, по сравнению с рассмотренными выше. Умножать можно только матрицы, отвечающие следующему требованию: **количество столбцов первой матрицы должно быть равно числу строк второй матрицы**.

Для простоты запоминания этого правила можно использовать диаграмму умножения, представленную на рисунке 1.

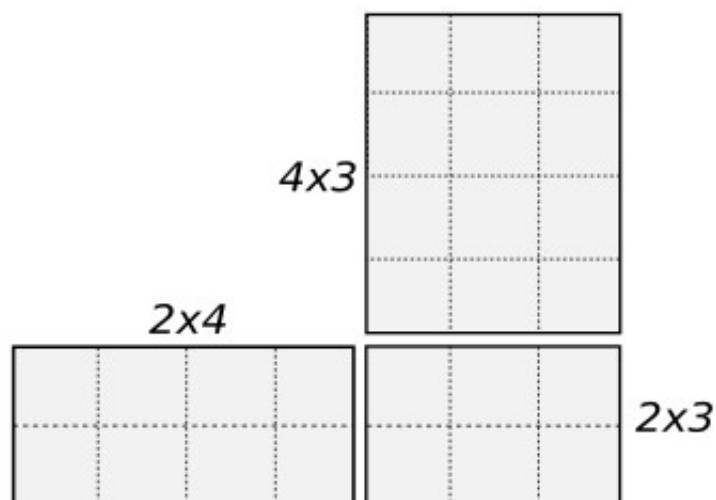


Рисунок 1.1 — Диаграмма матричного умножения

Рассмотрим умножение матриц на примере.

➤ Численный пример

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, B = \begin{pmatrix} 7 & 8 \\ 9 & 1 \\ 2 & 3 \end{pmatrix},$$

$$C = A \times B,$$

$$C = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 1 \\ 2 & 3 \end{pmatrix} =$$
$$= \begin{pmatrix} 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 2 & 1 \cdot 8 + 2 \cdot 1 + 3 \cdot 3 \\ 4 \cdot 7 + 5 \cdot 9 + 6 \cdot 2 & 4 \cdot 8 + 5 \cdot 1 + 6 \cdot 3 \end{pmatrix} = \begin{pmatrix} 31 & 19 \\ 85 & 55 \end{pmatrix}.$$

Каждый элемент c_{ij} новой матрицы является суммой произведений элементов i -ой строки первой матрицы и j -го столбца второй матрицы. Математически это записывается так:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{pmatrix},$$

$$C = A \times B,$$

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nk} \end{pmatrix},$$

$$c_{ij} = \sum_{r=1}^n a_{ir} \times b_{rj}.$$

➤ Пример на Python

Решим задачу умножения матриц на языке *Python*. Для этого будем использовать функцию **dot()** из библиотеки *Numpy*:

```
>>> A = np.matrix('1 2 3; 4 5 6')
>>> B = np.matrix('7 8; 9 1; 2 3')
>>> C = A.dot(B)
>>> print(C)
[[31 19]
 [85 55]]
```


Ниже представлены свойства произведения матриц. Примеры свойств будут показаны для квадратной матрицы.

Свойство 1. Ассоциативность умножения. Результат умножения матриц не зависит от порядка, в котором будет выполняться эта операция:

$$A \times (B \times C) = (A \times B) \times C.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \left(\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 7 & 8 \end{pmatrix} \right) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 52 & 68 \\ 70 & 92 \end{pmatrix} = \begin{pmatrix} 192 & 252 \\ 436 & 572 \end{pmatrix},$$
$$\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right) \times \begin{pmatrix} 2 & 4 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 192 & 252 \\ 436 & 572 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> C = np.matrix('2 4; 7 8')
>>> L = A.dot(B.dot(C))
>>> R = (A.dot(B)).dot(C)
>>> print(L)
[[192 252]
 [436 572]]
>>> print(R)
[[192 252]
 [436 572]]
```

Свойство 2. Дистрибутивность умножения. Произведение матрицы на сумму матриц равно сумме произведений матриц:

$$A \times (B + C) = A \times B + A \times C.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \left(\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} + \begin{pmatrix} 2 & 4 \\ 7 & 8 \end{pmatrix} \right) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 7 & 10 \\ 14 & 16 \end{pmatrix} = \begin{pmatrix} 35 & 42 \\ 77 & 94 \end{pmatrix},$$
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} + \begin{pmatrix} 16 & 20 \\ 34 & 44 \end{pmatrix} = \begin{pmatrix} 35 & 42 \\ 77 & 94 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> C = np.matrix('2 4; 7 8')
>>> L = A.dot(B + C)
>>> R = A.dot(B) + A.dot(C)
>>> print(L)
[[35 42]
 [77 94]]
>>> print(R)
[[35 42]
 [77 94]]
```

Свойство 3. Умножение матриц в общем виде не коммутативно. Это означает, что для матриц не выполняется правило независимости произведения от перестановки множителей:

$$A \times B \neq B \times A.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix},$$
$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 23 & 34 \\ 31 & 46 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> B = np.matrix('5 6; 7 8')
>>> L = A.dot(B)
>>> R = B.dot(A)
>>> print(L)
[[19 22]
 [43 50]]
>>> print(R)
[[23 34]
 [31 46]]
```

Свойство 4. Произведение заданной матрицы на единичную равно исходной матрице:

$$E \times A = A \times E = A.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},$$

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> E = np.matrix('1 0; 0 1')
>>> L = E.dot(A)
>>> R = A.dot(E)
>>> print(L)
[[1 2]
 [3 4]]
>>> print(R)
[[1 2]
 [3 4]]
>>> print(A)
[[1 2]
 [3 4]]
```

Свойство 5. Произведение заданной матрицы на нулевую матрицу равно нулевой матрице:

$$Z \times A = A \times Z = Z.$$

➤ Численный пример

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('1 2; 3 4')
>>> Z = np.matrix('0 0; 0 0')
>>> L = Z.dot(A)
>>> R = A.dot(Z)
>>> print(L)
[[0 0]
 [0 0]]
```

```
>>> print(R)
[[0 0]
 [0 0]]
>>> print(Z)
[[0 0]
 [0 0]]
```

1.5 Определитель матрицы

Определитель матрицы размера $n \times n$ (n -го порядка) является одной из ее численных характеристик. Определитель матрицы A обозначается как $|A|$ или $\det(A)$, его также называют **детерминантом**. Рассмотрим квадратную матрицу 2×2 в общем виде:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

Определитель такой матрицы вычисляется следующим образом:

$$|A| = \det(A) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} \times a_{22} - a_{12} \times a_{21}.$$

➤ Численный пример

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},$$

$$|A| = \det(A) = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = 1 \times 4 - 2 \times 3 = -2.$$

Перед тем, как привести методику расчета определителя в общем виде, введем понятие минора элемента определителя. *Минор элемента определителя* - это определитель, полученный из данного, путем вычеркивания всех элементов строки и столбца, на пересечении которых стоит данный элемент.

Для матрицы 3×3 следующего вида:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

Минор M_{23} будет выглядеть так:

$$M_{23} = \begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix}.$$

Введем еще одно понятие - *алгебраическое дополнение элемента определителя* - это минор этого элемента, взятый со знаком *плюс* или *минус*:

$$A_{ij} = (-1)^{i+j} M_{ij}.$$

В общем виде вычислить определитель матрицы можно через разложение определителя по элементам строки или столбца. Суть в том, что определитель равен сумме произведений элементов любой строки или столбца на их алгебраические дополнения.

Для матрицы 3×3 это правило будет выполняться следующим образом:

$$\begin{aligned} |A| &= \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \cdot A_{11} + a_{12} \cdot A_{12} + a_{13} \cdot A_{13} = \\ &= a_{11} \cdot (-1)^{1+1} M_{11} + a_{12} \cdot (-1)^{1+2} M_{12} + a_{13} \cdot (-1)^{1+3} M_{13} = \\ &= a_{11} \cdot \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{12} \cdot \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \cdot \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}. \end{aligned}$$

Это правило распространяется на матрицы любой размерности.

➤ Численный пример

$$\begin{aligned} |A| &= \begin{vmatrix} -4 & -1 & 2 \\ 10 & 4 & -1 \\ 8 & 3 & 1 \end{vmatrix} = (-4) \cdot A_{11} + (-1) \cdot A_{12} + 2 \cdot A_{13} = \\ &= (-4) \cdot (-1)^{1+1} M_{11} + (-1) \cdot (-1)^{1+2} M_{12} + 2 \cdot (-1)^{1+3} M_{13} = \\ &= (-4) \cdot \begin{vmatrix} 4 & -1 \\ 3 & 1 \end{vmatrix} - (-1) \cdot \begin{vmatrix} 10 & -1 \\ 8 & 1 \end{vmatrix} + 2 \cdot \begin{vmatrix} 10 & 4 \\ 8 & 3 \end{vmatrix} = \\ &= (-4) \cdot (4 + 3) + (10 + 8) + 2 \cdot (30 - 32) = \\ &= -28 + 18 - 4 = -14. \end{aligned}$$

➤ Пример на Python

На *Python* определитель посчитать очень просто. Создадим матрицу A размера 3×3 из приведенного выше численного примера:

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
```

Для вычисления определителя этой матрицы воспользуемся функцией **det()** из пакета **linalg**.

```
>>> np.linalg.det(A)
-14.000000000000009
```

Мы уже говорили про особенность работы *Python* с числами с плавающей точкой, поэтому можете полученное значение округлить до -14 .

Рассмотрим свойства определителя матрицы.

Свойство 1. Определитель матрицы остается неизменным при ее транспонировании:

$$\det(A) = \det(A^T).$$

➤ Пример на Python

Для округления чисел будем использовать функцию **round()**.

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> print(A.T)
[[-4 10  8]
 [-1  4  3]
 [ 2 -1  1]]
>>> det_A = round(np.linalg.det(A), 3)
>>> det_A_t = round(np.linalg.det(A.T), 3)
>>> print(det_A)
-14.0
>>> print(det_A_t)
-14.0
```

Свойство 2. Если у матрицы есть строка или столбец, состоящие из нулей, то определитель такой матрицы равен нулю:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = 0.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 0 0 0; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [ 0  0  0]
 [ 8  3  1]]
>>> np.linalg.det(A)
0.0
```

Свойство 3. При перестановке строк матрицы знак ее определителя меняется на противоположный:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}; \quad A' = \begin{pmatrix} a_{21} & a_{22} & \dots & a_{2n} \\ a_{11} & a_{12} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix},$$

$$\det(A) = -\det(A').$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> B = np.matrix('10 4 -1; -4 -1 2; 8 3 1')
>>> print(B)
[[10  4 -1]
 [-4 -1  2]
 [ 8  3  1]]
>>> round(np.linalg.det(A), 3)
-14.0
>>> round(np.linalg.det(B), 3)
14.0
```


Свойство 4. Если у матрицы есть две одинаковые строки, то ее определитель равен нулю:

$$\begin{vmatrix} a_1 & a_2 & \dots & a_n \\ a_1 & a_2 & \dots & a_n \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = 0.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; -4 -1 2; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [-4 -1  2]
 [ 8  3  1]]
>>> np.linalg.det(A)
0.0
```

Свойство 5. Если все элементы строки или столбца матрицы умножить на какое-то число, то и определитель будет умножен на это число:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \lambda \cdot a_{21} & \lambda \cdot a_{22} & \dots & \lambda \cdot a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \lambda \cdot \det(A).$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> k = 2
>>> B = A.copy()
>>> B[2, :] = k * B[2, :]
>>> print(B)
[[-4 -1  2]
 [10  4 -1]
 [16  6  2]]
>>> det_A = round(np.linalg.det(A), 3)
>>> det_B = round(np.linalg.det(B), 3)
>>> det_A * k
-28.0
>>> det_B
-28.0
```

Свойство 6. Если все элементы строки или столбца можно представить как сумму двух слагаемых, то определитель такой матрицы равен сумме определителей двух соответствующих матриц:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} + \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; -4 -1 2; 8 3 1')
>>> B = np.matrix('-4 -1 2; 8 3 2; 8 3 1')
>>> C = A.copy()
>>> C[1, :] += B[1, :]
>>> print(C)
[[-4 -1  2]
 [ 4  2  4]
 [ 8  3  1]]
>>> print(A)
[[-4 -1  2]
 [-4 -1  2]
 [ 8  3  1]]
>>> print(B)
[[-4 -1  2]
 [ 8  3  2]
 [ 8  3  1]]
>>> round(np.linalg.det(C), 3)
4.0
>>> round(np.linalg.det(A), 3) + round(np.linalg.det(B), 3)
4.0
```

Свойство 7. Если к элементам одной строки прибавить элементы другой строки умноженные на одно и тоже число, то определитель матрицы не изменится:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} + \beta \cdot a_{11} & a_{22} + \beta \cdot a_{12} & \dots & a_{2n} + \beta \cdot a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> k = 2
>>> B = A.copy()
>>> B[1, :] = B[1, :] + k * B[0, :]
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> print(B)
[[-4 -1  2]
 [ 2  2  3]
 [ 8  3  1]]
>>> round(np.linalg.det(A), 3)
-14.0
>>> round(np.linalg.det(B), 3)
-14.0
```

Свойство 8. Если строка или столбец матрицы является линейной комбинацией других строк (столбцов), то определитель такой матрицы равен нулю:

$$a_{2i} = \alpha \cdot a_{1i} + \beta \cdot a_{3i}; \quad \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = 0.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> k = 2
>>> A[1, :] = A[0, :] + k * A[2, :]
>>> round(np.linalg.det(A), 3)
0.0
```

Свойство 9. Если матрица содержит пропорциональные строки, то ее определитель равен нулю:

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \beta \cdot a_{11} & \beta \cdot a_{12} & \dots & \beta \cdot a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = 0.$$

➤ Пример на Python

```
>>> A = np.matrix('-4 -1 2; 10 4 -1; 8 3 1')
>>> print(A)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> k = 2
>>> A[1, :] = k * A[0, :]
>>> print(A)
[[-4 -1  2]
 [-8 -2  4]
 [ 8  3  1]]
>>> round(np.linalg.det(A), 3)
0.0
```

1.6 Обратная матрица

Обратной матрицей A^{-1} матрицы A называют матрицу, удовлетворяющую следующему равенству:

$$A \times A^{-1} = A^{-1} \times A = E,$$

где E - это единичная матрица.

Для того, чтобы у квадратной матрицы A была обратная матрица необходимо и достаточно чтобы определитель $|A|$ был не равен нулю.

Введем понятие **союзной матрицы**. **Союзная матрица** A^* строится на базе исходной A путем замены всех элементов матрицы A на их алгебраические дополнения.

Исходная матрица:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Союзная ей матрица A^* :

$$A^* = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix}.$$

Транспонируя матрицу A^* , мы получим так называемую **присоединенную матрицу** A^{*T} :

$$A^{*T} = \begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \dots & \dots & \dots & \dots \\ A_{1n} & A_{2n} & \dots & A_{nn} \end{pmatrix}.$$

Теперь, зная как вычислять определитель и присоединенную матрицу, мы можем определить матрицу A^{-1} , обратную матрице A :

$$A^{-1} = \frac{1}{\det(A)} \times A^{*T}.$$

➤ Численный пример

Пример вычисления обратной матрицы. Пусть дана исходная матрица A , следующего вида:

$$A = \begin{pmatrix} 1 & -3 \\ 2 & 5 \end{pmatrix}.$$

Для начала найдем определитель матрицы A :

$$|A| = \begin{vmatrix} 1 & -3 \\ 2 & 5 \end{vmatrix} = 5 - (-6) = 11.$$

Как видно из приведенных вычислений, определитель матрицы не равен нулю, значит у матрицы A есть обратная. Построим присоединенную матрицу, для этого вычислим алгебраические дополнения для каждого элемента матрицы A :

$$A_{11} = (-1)^2 \cdot M_{11} = (-1)^2 \cdot 5 = 5,$$

$$A_{12} = (-1)^3 \cdot M_{12} = (-1)^3 \cdot 2 = -2,$$

$$A_{21} = (-1)^3 \cdot M_{21} = (-1)^3 \cdot (-3) = 3,$$

$$A_{22} = (-1)^4 \cdot M_{22} = (-1)^4 \cdot 1 = 1.$$

Союзная матрица будет иметь следующий вид:

$$A^* = \begin{pmatrix} 5 & -2 \\ 3 & 1 \end{pmatrix}.$$

Присоединенная матрица получается из союзной путем транспонирования:

$$A^{*T} = \begin{pmatrix} 5 & 3 \\ -2 & 1 \end{pmatrix}.$$

Находим обратную матрицу:

$$A^{-1} = \frac{1}{\det(A)} \cdot A^{*T} = \frac{1}{11} \cdot \begin{pmatrix} 5 & 3 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} \frac{5}{11} & \frac{3}{11} \\ -\frac{2}{11} & \frac{1}{11} \end{pmatrix}$$

➤ Пример на Python

Решим задачу определения обратной матрицы на *Python*. Для получения обратной матрицы будем использовать функцию *inv()*:

```
>>> A = np.matrix('1 -3; 2 5')
>>> A_inv = np.linalg.inv(A)
>>> print(A_inv)
[[ 0.45454545  0.27272727]
 [-0.18181818  0.09090909]]
```

Рассмотрим свойства обратной матрицы.

Свойство 1. Обратная матрица обратной матрицы есть исходная матрица:

$$(A^{-1})^{-1} = A.$$

➤ Пример на Python

```
>>> A = np.matrix('1. -3.; 2. 5.')
>>> A_inv = np.linalg.inv(A)
>>> A_inv_inv = np.linalg.inv(A_inv)
>>> print(A)
[[1. -3.]
 [2.  5.]]
>>> print(A_inv_inv)
[[1. -3.]
 [2.  5.]]
```

Свойство 2. Обратная матрица транспонированной матрицы равна транспонированной матрице от обратной матрицы:

$$(A^T)^{-1} = (A^{-1})^T.$$

➤ Пример на Python

```
>>> A = np.matrix('1. -3.; 2. 5.')
>>> L = np.linalg.inv(A.T)
>>> R = (np.linalg.inv(A)).T
>>> print(L)
[[ 0.45454545 -0.18181818]
 [ 0.27272727  0.09090909]]
>>> print(R)
[[ 0.45454545 -0.18181818]
 [ 0.27272727  0.09090909]]
```

Свойство 3. Обратная матрица произведения матриц равна произведению обратных матриц:

$$(A_1 \times A_2)^{-1} = A_2^{-1} \times A_1^{-1}.$$

➤ Пример на Python

```
>>> A = np.matrix('1. -3.; 2. 5.')
>>> B = np.matrix('7. 6.; 1. 8.')
>>> L = np.linalg.inv(A.dot(B))
>>> R = np.linalg.inv(B).dot(np.linalg.inv(A))
>>> print(L)
[[ 0.09454545  0.03272727]
 [-0.03454545  0.00727273]]
>>> print(R)
[[ 0.09454545  0.03272727]
 [-0.03454545  0.00727273]]
```

1.7 Ранг матрицы

Ранг матрицы является еще одной важной численной характеристикой. Рангом называют максимальное число линейно независимых строк (столбцов) матрицы. Линейная независимость означает, что строки (столбцы) не могут быть линейно выражены через другие строки (столбцы). Ранг матрицы можно найти через ее миноры, он равен наибольшему порядку минора, который не равен нулю. Существование ранга у матрицы не зависит от того квадратная она или нет.

Вычислим ранг матрицы с помощью *Python*. Создадим единичную матрицу:

```
>>> m_eye = np.eye(4)
>>> print(m_eye)
[[1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  1.  0.]
 [0.  0.  0.  1.]]
```

Ранг такой матрицы равен количеству ее столбцов (или строк), в нашем случае ранг будет равен четырем, для его вычисления на *Python* воспользуемся функцией ***matrix_rank()***:

```
>>> rank = np.linalg.matrix_rank(m_eye)
>>> print(rank)
4
```

Если мы приравняем элемент в нижнем правом углу к *нулю*, то ранг станет равен трем:

```
>>> m_eye[3][3] = 0
>>> print(m_eye)
[[1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  1.  0.]
 [0.  0.  0.  0.]]
>>> rank = np.linalg.matrix_rank(m_eye)
>>> print(rank)
3
```


Глава 2. Решение систем линейных уравнений

Системой линейных уравнений называют совокупность уравнений следующего вида:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \end{cases}.$$

Решить такую систему - это значит найти n чисел, при подстановке которых уравнение превращается в тождество. Если все $b_k = 0$, то такая система называется **однородной**.

Если хотя бы одно b_k не равно нулю, то система называется **неоднородной**. Если система имеет хотя бы одно решение, то она называется **совместной**, если решений нет, то **несовместной**.

2.1 Решение неоднородной системы в матричной форме

Запишем неоднородную систему линейных уравнений в матричной форме, для этого выделим **главную матрицу системы**, которая состоит из коэффициентов перед неизвестными в левой части, **матрицу столбец из элементов правой части** и **матрицу столбец из неизвестных**.

Главная матрица системы:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

Матрица столбец из элементов правых частей:

$$B = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix}.$$

Матрица столбец из неизвестных:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}.$$

Используя правило умножения матриц, можно составить систему линейных уравнений в матричной форме:

$$A \times X = B.$$

Решить систему в матричной форме - это значит по известным матрицам A и B найти матрицу X . Матрицу X можно найти через умножение матрицы обратной матрице A и матрицы B :

$$X = A^{-1} \times B.$$

Если к элементам главной матрицы системы добавить матрицу столбец из элементов правой части, то получим расширенную матрицу системы:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{pmatrix}.$$

Если ранг расширенной матрицы равен рангу главной матрицы системы, то такая система уравнений будет **совместной**, в противном случае - **несовместной**. Совместная **система имеет одно решение, если ее ранг равен числу неизвестных**, если ранг меньше числа неизвестных, то число решений **бесконечное множество**.

➤ Численный пример

Решим матричным методом следующую систему:

$$\begin{cases} 3x - y + 2z = -4 \\ x + 4y - z = 10 \\ 2x + 3y + z = 8 \end{cases}.$$

Для этого построим матрицы системы:

$$A = \begin{pmatrix} 3 & -1 & 2 \\ 1 & 4 & -1 \\ 2 & 3 & 1 \end{pmatrix}, X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, B = \begin{pmatrix} -4 \\ 10 \\ 8 \end{pmatrix}.$$

Найдем определитель матрицы A :

$$A = \begin{vmatrix} 3 & -1 & 2 \\ 1 & 4 & -1 \\ 2 & 3 & 1 \end{vmatrix} = 14.$$

Построим присоединенную матрицу для матрицы A :

$$A^{*T} = \begin{pmatrix} 7 & 7 & -7 \\ 7 & -1 & -11 \\ -7 & 5 & 13 \end{pmatrix}.$$

Обратная матрица имеет следующий вид:

$$A^{-1} = \frac{1}{|A|} \cdot A^{*T} = \frac{1}{14} \cdot \begin{pmatrix} 7 & 7 & -7 \\ 7 & -1 & -11 \\ -7 & 5 & 13 \end{pmatrix} = \begin{pmatrix} \frac{7}{14} & \frac{7}{14} & \frac{-7}{14} \\ \frac{7}{14} & \frac{-1}{14} & \frac{-11}{14} \\ \frac{-7}{14} & \frac{5}{14} & \frac{13}{14} \end{pmatrix}.$$

Найдем матрицу X :

$$\begin{aligned} X = A^{-1} \times B &= \begin{pmatrix} \frac{7}{14} & \frac{7}{14} & \frac{-7}{14} \\ \frac{7}{14} & \frac{-1}{14} & \frac{-11}{14} \\ \frac{-7}{14} & \frac{5}{14} & \frac{13}{14} \end{pmatrix} \times \begin{pmatrix} -4 \\ 10 \\ 8 \end{pmatrix} = \begin{pmatrix} -\frac{28}{14} + \frac{70}{14} - \frac{56}{14} \\ \frac{12}{14} - \frac{10}{14} + \frac{40}{14} \\ \frac{20}{14} - \frac{110}{14} + \frac{104}{14} \end{pmatrix} = \\ &= \begin{pmatrix} -1 \\ 3 \\ 1 \end{pmatrix}. \end{aligned}$$

Для проверки полученного решения подставим его в исходные уравнения:

$$\begin{cases} 3 \times (-1) - 3 + 2 \times 1 = -4 \\ (-1) + 4 \times 3 - 1 = 10 \\ 2 \times (-1) + 3 \times 3 + 1 = 8 \end{cases} = \begin{cases} -4 = -4 \\ 10 = 10 \\ 8 = 8 \end{cases}.$$

Как видно, найденное решение превратило уравнения в тождества, значит решение верное. Решим систему, используя предложенный метод на *Python*.

➤ Пример на Python

```
>>> A = np.matrix('3 -1 2; 1 4 -1; 2 3 1')
>>> B = np.matrix('-4; 10; 8')
>>> print(A)
[[3 -1  2]
 [1  4 -1]
 [2  3  1]]
>>> print(B)
[[-4]
 [10]
 [ 8]]
>>> A_inv = np.linalg.inv(A)
>>> print(A_inv)
[[0.5  0.5 -0.5]
 [-0.21428571 -0.07142857  0.35714286]
 [-0.35714286 -0.78571429  0.92857143]]
>>> X = A_inv.dot(B)
>>> print(X)
[[-1.]
 [ 3.]
 [ 1.]]
```

2.2 Метод Крамера

Метод Крамера можно использовать для решения неоднородных систем линейных уравнений, при этом **определитель главной матрицы системы должен быть отличен от нуля**. Полученное данным методом решение будет единственным.

Построим матрицы X_1, X_1, \dots, X_n , которые получаются из главной матрицы системы путем замены столбца с индексом n на матрицу-столбец правых частей.

Исходные данные. Главная матрица системы:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Матрица столбец из элементов правых частей:

$$B = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}.$$

Матрицы X_1, X_1, \dots, X_n будут выглядеть так:

$$X_1 = \begin{pmatrix} b_1 & a_{12} & \dots & a_{1n} \\ b_2 & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ b_n & a_{n2} & \dots & a_{nn} \end{pmatrix}, \quad X_2 = \begin{pmatrix} a_{11} & b_1 & \dots & a_{1n} \\ a_{21} & b_2 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & b_n & \dots & a_{nn} \end{pmatrix}, \quad \dots,$$

$$X_n = \begin{pmatrix} a_{11} & a_{12} & \dots & b_1 \\ a_{21} & a_{22} & \dots & b_2 \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & b_n \end{pmatrix}.$$

Неизвестные x_1, x_2, \dots, x_n вычисляются с помощью формул Крамера:

$$x_1 = \frac{|X_1|}{|A|},$$

$$x_2 = \frac{|X_2|}{|A|},$$

...

$$x_n = \frac{|X_n|}{|A|},$$

где $|A|$ - это определитель главной матрицы системы;

$|X_n|$ - определитель матрицы X_n , полученный из главной путем замены n -го столбца на матрицу B .

Решим систему линейных уравнений из раздела про матричный метод, но в данном случае, методом Крамера.

> Численный пример

Дана система уравнений:

$$\begin{cases} 3x - y + 2z = -4 \\ x + 4y - z = 10 \\ 2x + 3y + z = 8 \end{cases}.$$

Матрицы системы имеют вид:

$$A = \begin{pmatrix} 3 & -1 & 2 \\ 1 & 4 & -1 \\ 2 & 3 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad B = \begin{pmatrix} -4 \\ 10 \\ 8 \end{pmatrix}.$$

Найдем определитель матрицы A :

$$A = \begin{vmatrix} 3 & -1 & 2 \\ 1 & 4 & -1 \\ 2 & 3 & 1 \end{vmatrix} = 14.$$

Формулы Крамера для поиска неизвестных x , y и z будут выглядеть так:

$$x = \frac{|X|}{|A|},$$

$$y = \frac{|Y|}{|A|},$$

$$z = \frac{|Z|}{|A|}.$$

Найдем определители матриц X , Y , Z :

$$|X| = \begin{vmatrix} -4 & -1 & 2 \\ 10 & 4 & -1 \\ 8 & 3 & 1 \end{vmatrix} = -14,$$

$$|Y| = \begin{vmatrix} 3 & -4 & 2 \\ 1 & 10 & -1 \\ 2 & 8 & 1 \end{vmatrix} = 42,$$

$$|Z| = \begin{vmatrix} 3 & -1 & -4 \\ 1 & 4 & 10 \\ 2 & 3 & 8 \end{vmatrix} = 14.$$

Найдем численные значения неизвестных:

$$x = \frac{-14}{14} = -1,$$

$$y = \frac{42}{14} = 3,$$

$$z = \frac{14}{14} = 1.$$

Полученное решение совпадает с тем, что было найдено в предыдущем разделе.

➤ Пример на Python

Решим систему методом Крамера на *Python*:

```
>>> A = np.matrix('3 -1 2; 1 4 -1; 2 3 1')
>>> print(A)
[[ 3 -1  2]
 [ 1  4 -1]
 [ 2  3  1]]
>>> B = np.matrix('-4; 10; 8')
>>> print(B)
[[-4]
 [10]
 [ 8]]
>>> A_det = np.linalg.det(A)
>>> print(A_det)
14.0
>>> X_m = np.matrix(A)
>>> X_m[:, 0] = B
>>> print(X_m)
[[-4 -1  2]
 [10  4 -1]
 [ 8  3  1]]
>>> Y_m = np.matrix(A)
>>> Y_m[:, 1] = B
>>> print(Y_m)
[[ 3 -4  2]
 [ 1 10 -1]
 [ 2  8  1]]
```

```

>>> Z_m = np.matrix(A)
>>> Z_m[:, 2] = B
>>> print(Z_m)
[[ 3 -1 -4]
 [ 1  4 10]
 [ 2  3  8]]
>>> x = np.linalg.det(X_m) / A_det
>>> y = np.linalg.det(Y_m) / A_det
>>> z = np.linalg.det(Z_m) / A_det
>>> print(x)
-1.0
>>> print(y)
3.0
>>> print(z)
1.0

```

2.3 Метод Гаусса

Метод Гаусса отличается от рассмотренных нами выше, он более удобен при ручном решении систем линейных уравнений. Суть его заключается в том, что мы приводим систему уравнений к ступенчатому виду, в идеале, у нас из системы вида¹:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases},$$

должна получиться следующая система:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ \phantom{a_{11}x_1 +} a'_{22}x_2 + \dots + a'_{2n}x_n = b_2 \\ \phantom{a_{11}x_1 +} \phantom{a'_{22}x_2 +} \dots \\ \phantom{a_{11}x_1 +} \phantom{a'_{22}x_2 +} a''_{nn}x_n = b_n \end{cases}.$$

Из нее, последовательно выражая неизвестные, получим их численные значения.

➤ Численный пример

Рассмотрим работу метода Гаусса на уже знакомой нам системе уравнений:

$$\begin{cases} 3x - y + 2z = -4 \\ x + 4y - z = 10 \\ 2x + 3y + z = 8 \end{cases}.$$

1 Количество неизвестных обязательно должно совпадать с числом уравнений в системе

Для начала поменяем строки 1 и 3 местами:

$$\begin{cases} 2x + 3y + z = 8 \\ x + 4y - z = 10 \\ 3x - y + 2z = -4 \end{cases}.$$

Разделим первую строку на 2 и вычтем из нее вторую строку, после этого умножим первую строку на $\frac{3}{2}$ и вычтем из нее третью строку, полученные уравнения подставляем во вторую и третью строку соответственно:

$$\begin{cases} 2x + 3y + z = 8 \\ -\frac{5}{2}y + \frac{3}{2}z = -6 \\ \frac{11}{2}y - \frac{1}{2}z = 16 \end{cases}$$

Умножим вторую строку на $(-11/5)$ и вычтем из нее третью строку:

$$\begin{cases} 2x + 3y + z = 8 \\ -\frac{5}{2}y + \frac{3}{2}z = -6 \\ -\frac{28}{10}z = \frac{-14}{5} \end{cases}$$

Из полученной системы мы можем определить значение неизвестной z :

$$z = 1.$$

Подставляя $z = 1$ во второе уравнение, получим значение y :

$$y = 3.$$

Теперь мы можем вычислить x :

$$x = -1.$$

Полученные значения совпадают с решением из предыдущих разделов.

2.4 Решение системы линейных уравнений на *Python*

Отдельно рассмотрим инструмент, который предоставляет *Numpy* для решения систем линейных уравнений. Создадим матрицы, так, как это было сделано в разделе 2.1 “Решение неоднородной системы в матричной форме”:

```
>>> A = np.matrix('3 -1 2; 1 4 -1; 2 3 1')
>>> B = np.matrix('-4; 10; 8')
>>> print(A)
[[ 3 -1  2]
 [ 1  4 -1]
 [ 2  3  1]]
>>> print(B)
[[-4]
 [10]
 [ 8]]
```

Для решения системы воспользуемся методом ***solve()*** из пакета ***linalg***:

```
>>> X = np.linalg.solve(A, B)
>>> print(X)
[[-1.]
 [ 3.]
 [ 1.]]
```

Глава 3. Основы векторной алгебры

3.1 Векторные и скалярные величины

В рамках геометрической интерпретации под вектором понимается направленный отрезок прямой, характеризуемый длиной и направлением (см. рисунок 3.1). Если вектор рассматривать как алгебраический объект, то он может быть представлен в виде строки или столбца чисел, то есть в виде матрицы размера $(1, n)$ либо $(n, 1)^2$. Связь между геометрическим и алгебраическим представлением более подробно раскрыта в разделе 3.3 “Координаты вектора”.



Рисунок 3.1 - Графическое представление вектора

Геометрическая интерпретация позволяет проще воспринимать понятие вектора, она также удобна при решении задач из курса физики. Далее мы будем обращаться к геометрическому представлению с целью визуализации и более глубокого интуитивного понимания рассматриваемого вопроса.

В физике и технических дисциплинах (механика, электротехника и т.п.) различают скалярные и векторные величины. **Скалярной называется величина, которая характеризуется только числовым значением. Векторной называется величина, которая характеризуется числовым значением и направлением.**

Вектор будем обозначать одной маленькой латинской буквой полужирного начертания, вот так: ***a***, либо двумя большими латинскими буквами, первая обозначает точку начала, вторая - конец вектора: ***AB***.

Длина вектора (эту величину еще называют **модуль вектора**) изображается так: **$|a|$**

2 Для обозначения размерности вектора удобно использовать запись $(1, n)$ вместо $1 \times n$

3.2 Проекция вектора на ось

Проекция вектора на ось равна произведению модуля вектора на косинус угла между вектором и осью.

Рассмотрим вектор \mathbf{a} и ось k . Проекцией вектора \mathbf{a} на ось k будет произведение $|\mathbf{a}|$ и косинуса угла φ между вектором \mathbf{a} и осью k (см. рисунок 3.2).

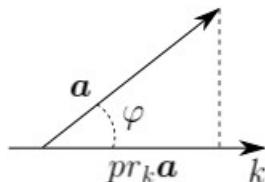


Рисунок 3.2 - Проекция вектора на ось

В виде формулы это можно записать так:

$$pr_k \mathbf{a} = |\mathbf{a}| \cdot \cos(\varphi)$$

3.3 Координаты вектора

Координатами вектора называют его проекции на оси координат:

$$\mathbf{a} = (a_x, a_y, a_z),$$

$$pr_{ox} \mathbf{a} = a_x,$$

$$pr_{oy} \mathbf{a} = a_y,$$

$$pr_{oz} \mathbf{a} = a_z.$$

Если известны координаты начальной и конечной точки, то координаты вектора определяются следующим образом:

$$a_x = x_2 - x_1,$$

$$a_y = y_2 - y_1,$$

$$a_z = z_2 - z_1.$$

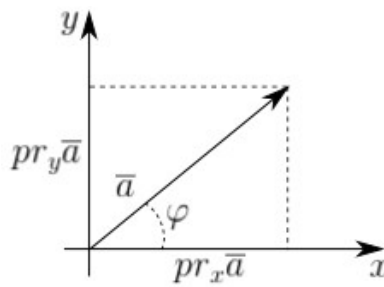


Рисунок 3.3 Проекция вектора на оси координат

➤ Численный пример

Рассмотрим вектор AB , у которого начальная точка имеет координаты $A(1, 2, 3)$, конечная - $B(7, 9, 15)$. Тогда вектор AB , выраженный через координаты проекций на оси координат будет выглядеть так:

$$A(1, 2, 3),$$

$$B(7, 9, 5),$$

$$AB = (7 - 1, 9 - 2, 5 - 3) \rightarrow AB = (6, 7, 2).$$

➤ Пример на Python

Теперь решим эту задачу на языке *Python*. Как уже было сказано выше, вектор может быть задан через координаты его начала и конца, в *Python* для этого будем использовать *Numpy* массивы. Зададим эти точки в виде переменных:

```
>>> A = np.array([1, 2, 3])
>>> print(A)
[1 2 3]
>>> B = np.array([3, 5, 2])
>>> print(B)
[3 5 2]
>>> AB = B - A
>>> print(AB)
[ 2  3 -1]
```

3.4 Модуль вектора и его направление в пространстве

Модулем вектора называется величина, равная квадратному корню из суммы квадратов его координат:

$$\mathbf{a} = (a_x, a_y, a_z),$$
$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

Направление вектора определяется косинусами углов между вектором и осями координат:

α - угол между вектором \mathbf{a} и осью Ox .

β - угол между вектором \mathbf{a} и осью Oy .

γ - угол между вектором \mathbf{a} и осью Oz .

Направляющие векторы рассчитываются по следующим формулам:

$$\cos(\alpha) = \frac{a_x}{|\mathbf{a}|},$$

$$\cos(\beta) = \frac{a_y}{|\mathbf{a}|},$$

$$\cos(\gamma) = \frac{a_z}{|\mathbf{a}|}.$$

➤ Численный пример

Для демонстрации будем использовать вектор AB из раздела про координаты вектора:

$$AB = (6, 7, 2).$$

Найдем его модуль:

$$|AB| = \sqrt{a_x^2 + a_y^2 + a_z^2} = \sqrt{6^2 + 7^2 + 2^2} \approx 9.434.$$

Теперь мы можем вычислить направляющие косинусы:

$$\cos(\alpha) = \frac{a_x}{|\mathbf{a}|} = \frac{6}{9.434} \approx 0.636,$$

$$\cos(\beta) = \frac{a_y}{|\mathbf{a}|} = \frac{7}{9.434} \approx 0.742,$$

$$\cos(\gamma) = \frac{a_z}{|\mathbf{a}|} = \frac{2}{9.434} \approx 0.212.$$

➤ Пример на Python

Для вычисления модуля вектора будем использовать функцию ***norm()*** из библиотеки ***numpy.linalg***:

```
>>> AB=np.array([6, 7, 2])
>>> module_AB=np.linalg.norm(AB)
>>> print(module_AB)
9.433981132056603
```

Напишем функции для вычисления косинусов:

```
>>> cos_x = lambda a: a[0] / np.linalg.norm(a)
>>> cos_y = lambda a: a[1] / np.linalg.norm(a)
>>> cos_z = lambda a: a[2] / np.linalg.norm(a)
>>> cos_x(AB)
0.6359987280038161
>>> cos_y(AB)
0.741998516004452
>>> cos_z(AB)
0.211999576001272
```

Для того, чтобы получить значения косинусов в виде списка составим следующую функцию:

```
>>> vect_cos = lambda a: [a[i] / np.linalg.norm(a) for i in range(len(a))]
>>> vect_cos(AB)
[0.6359987280038161, 0.741998516004452, 0.211999576001272]
```

3.4 Операции над векторами

3.4.1 Сложение векторов

Геометрическое представление векторов прекрасно подходит для иллюстрации правила параллелограмма для сложения векторов (см. рисунок 3.4).

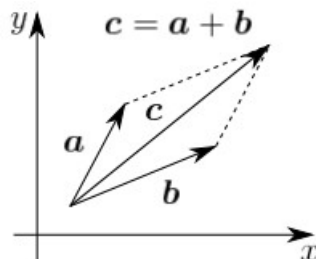


Рисунок 3.4 - Сложение векторов. Правило параллелограмма

Суть в том, что от конца первого вектора откладывается второй, а от конца второго - первый, в полученном на этих векторах параллелограмме проводится диагональ, которая и будет являться результирующим вектором. Это правило можно упростить: достаточно отложить от конца первого вектора второй и соединить начало первого и конец второго, такой подход называется **правилом треугольника** (иллюстрация приведена на рисунке 3.5).

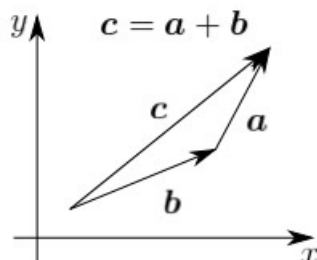


Рисунок 3.5 - Сложение векторов. Правило треугольника

Если рассматривать вектора в алгебраической форме - через численные значения их проекций, то при суммировании векторов, мы получаем вектор, проекции которого равны сумме проекций слагаемых векторов. Для двумерного вектора это будет выглядеть так: если у нас есть два вектора:

$$\mathbf{a} = (a_x, a_y),$$

$$\mathbf{b} = (b_x, b_y).$$

Вектор суммы данных векторов будет таким:

$$(\mathbf{a} + \mathbf{b}) = (a_x + b_x, a_y + b_y).$$

➤ Численный пример

Рассмотрим сложение векторов на примере. Пусть нам даны следующие вектора:

$$\mathbf{a} = (2, 1),$$

$$\mathbf{b} = (1, 3).$$

Вычислим вектор суммы:

$$(\mathbf{a} + \mathbf{b}) = (2 + 1, 1 + 3) \rightarrow (3, 4).$$

На рисунке 3.6 представлено суммирование векторов по правилу треугольника.

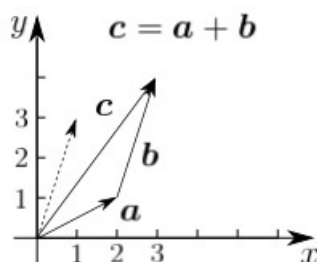


Рисунок 3.6 - Пример использования правила треугольника для сложения векторов

➤ Пример на Python

На *Python* задача суммирования векторов сводится к получению суммы соответствующих массивов:

```
>>> a = np.array([2, 1])
>>> b = np.array([1, 3])
>>> c = a + b
>>> print(c)
[3 4]
```

3.4.2 Вычитание векторов

Для **вычитания векторов** также существует свое **правило треугольника**. Графический вид этого правила представлен на рисунке 3.7.

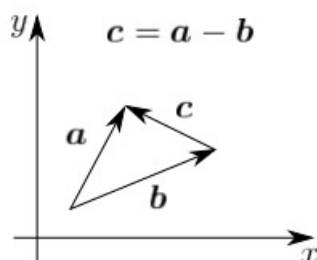


Рисунок 3.7 - Вычитание векторов. Правило треугольника

При вычитании, результирующий вектор выходит из конца второго вектора в конец первого вектора. Для двумерного случая с векторами:

$$\mathbf{a} = (a_x, a_y),$$

$$\mathbf{b} = (b_x, b_y).$$

Вектор разности определяется так:

$$(a - b) = (a_x - b_x, a_y - b_y).$$

➤ Численный пример

В качестве исходных возьмем вектора из примера про сложение:

$$a = (2, 1),$$

$$b = (1, 3).$$

Найдем разность вектора b и a :

$$(a - b) = (1 - 2, 3 - 1) \rightarrow (-1, 2).$$

Иллюстрация примера приведена на рисунке 3.8.

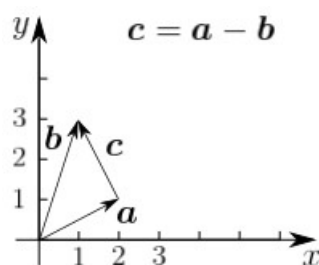


Рисунок 3.8 - Вычитание векторов. Пример

➤ Пример на Python

```
>>> a = np.array([2, 1])
>>> b = np.array([1, 3])
>>> c = b - a
>>> print(c)
[-1  2]
```

3.4.3 Умножение вектора на число

При **умножении вектора на число**, все компоненты вектора умножаются на это число. Графически умножение вектора на число показано на рисунке 3.9.

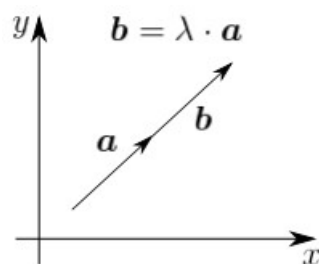


Рисунок 3.9 - Умножение вектора на число

Запишем алгебраическую формулу для данной операции:

$$\mathbf{a} = (a_x, a_y),$$

$$\lambda \cdot \mathbf{a} = (\lambda \cdot a_x, \lambda \cdot a_y).$$

➤ Численный пример

Умножим вектор \mathbf{a} с координатами $(2, 1)$ на число 2.

$$\mathbf{a} = (2, 1),$$

$$2 \cdot \mathbf{a} = (2 \cdot 2, 2 \cdot 1) \rightarrow (4, 2).$$

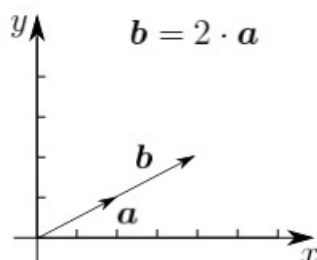


Рисунок 3.10 - Пример умножения вектора на число

➤ Пример на Python

```
>>> a = np.array([2, 1])
>>> k = 2
>>> c = k * a
>>> print(c)
[4 2]
```

3.4.4 Скалярное произведение двух векторов

Скалярным произведением двух векторов \mathbf{a} и \mathbf{b} называется число, равное произведению модулей этих векторов на косинус угла между ними:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos(\varphi).$$

Скалярное произведение можно выразить через координаты векторов:

$$\mathbf{a} = (a_x, a_y, a_z),$$

$$\mathbf{b} = (b_x, b_y, b_z),$$

$$\mathbf{a} \cdot \mathbf{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z.$$

Если транспонировать один из векторов, то скалярное произведение можно выразить через операцию умножения матриц:

$$\mathbf{a} \cdot \mathbf{b} = (a_x \ a_y \ a_z) \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z.$$

➤ Численный пример

Даны два вектора \mathbf{a} и \mathbf{b} :

$$\mathbf{a} = (2, 7, 3),$$

$$\mathbf{b} = (5, 9, 3).$$

Найдем их скалярное произведение:

$$\mathbf{a} \cdot \mathbf{b} = 2 \cdot 5 + 7 \cdot 9 + 3 \cdot 3 = 82.$$

➤ Пример на Python

Найдем скалярное произведение векторов:

```
>>> a = np.array([2, 7, 3])
>>> b = np.array([5, 9, 3])
>>> ab_scal = a.dot(b.T)
>>> ab_scal
82
```

3.4.5 Векторное произведение двух векторов

Векторным произведением двух векторов \mathbf{a} и \mathbf{b} называется третий вектор $\mathbf{c} = \mathbf{a} \times \mathbf{b}$, который удовлетворяет следующим условиям:

1. Вектор \mathbf{c} перпендикулярен векторам \mathbf{a} и \mathbf{b} .
2. Модуль вектора \mathbf{c} определяется по формуле:

$$|\mathbf{c}| = |\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \sin(\varphi).$$

3. Вектора \mathbf{a} , \mathbf{b} и \mathbf{c} образуют правую тройку векторов³.

Векторное произведение также можно выразить через координаты перемножаемых векторов:

$$\mathbf{a} = (a_x, a_y, a_z),$$

$$\mathbf{b} = (b_x, b_y, b_z),$$

3 <https://bit.ly/2uMXVxr>

$$\mathbf{c} = \mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}.$$

Векторное произведение равно определителю матрицы, составленной из единичных орт-векторов и проекций векторов \mathbf{a} и \mathbf{b} на данные орт-векторы.

> Численный пример

Для численного примера возьмем уже знакомые нам вектора \mathbf{a} и \mathbf{b} :

$$\mathbf{a} = (2, 7, 3),$$

$$\mathbf{b} = (5, 9, 3).$$

Найдем их модули:

$$|\mathbf{a}| = \sqrt{2^2 + 7^2 + 3^2} = 7.874,$$

$$|\mathbf{b}| = \sqrt{5^2 + 9^2 + 3^2} = 10.724.$$

Зная значение скалярного произведения из предыдущего параграфа, определим косинус угла между этими векторами:

$$\cos(\varphi) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \cdot |\mathbf{b}|} = \frac{82}{7.874 \cdot 10.724} = 0.971.$$

Синус угла можно вычислить следующим образом:

$$\sin(\varphi) = \sqrt{1 - \cos^2(\varphi)} = \sqrt{1 - 0.971^2} = 0.239.$$

Рассчитаем модуль векторного произведения:

$$|\mathbf{c}| = |\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \sin(\varphi) = 7.874 \cdot 10.724 \cdot 0.239 = 20.181.$$

Получим векторное произведение, выраженное через координаты векторов.

$$\begin{aligned} \mathbf{c} = \mathbf{a} \times \mathbf{b} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 2 & 7 & 3 \\ 5 & 9 & 3 \end{vmatrix} = \mathbf{i} \cdot \begin{vmatrix} 7 & 3 \\ 9 & 3 \end{vmatrix} - \mathbf{j} \cdot \begin{vmatrix} 2 & 3 \\ 5 & 3 \end{vmatrix} + \mathbf{k} \cdot \begin{vmatrix} 2 & 7 \\ 5 & 9 \end{vmatrix} = \\ &= -6 \cdot \mathbf{i} + 9 \cdot \mathbf{j} - 17 \cdot \mathbf{k}. \end{aligned}$$

➤ Пример на Python

Решим задачу векторного умножения на *Python*, для начала определимся с модулем. Будем работать с векторами из раздела про скалярное произведение:

```
>>> a = np.array([2, 7, 3])
>>> b = np.array([5, 9, 3])
```

Найдем косинус угла между векторами. Это можно сделать двумя способами:

- предварительно реализовав соответствующую формулу:

```
>>> vect_cos = lambda a, b: a.T.dot(b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

- воспользоваться уже готовой формулой из *scipy*, которая вычисляет косинусное расстояние:

```
>>> vect_cos(a, b)
0.97111114784134267
```

Используя полученную функцию, найдем синус угла между векторами:

```
>>> vect_sin = lambda a, b: (1 - vect_cos(a, b)**2)**0.5
>>> vect_sin(a, b)
0.23862626949623264
```

Теперь у нас есть все данные, чтобы построить формулу для модуля векторного произведения:

```
>>> prod_vect = lambda a, b: np.linalg.norm(a)*np.linalg.norm(b)*vect_sin(a, b)
>>> prod_vect(a, b)
20.149441679609886
```

3.4.6 Смешанное произведение трех векторов

Смешанное произведение трех векторов (его еще называют векторно-скалярным) определяется так: два вектора умножаются векторно, после этого, полученное произведение необходимо умножить на третий вектор скалярно.

Рассмотрим три вектора a , b и c :

$$\mathbf{a} = (a_x, a_y, a_z),$$

$$\mathbf{b} = (b_x, b_y, b_z),$$

$$\mathbf{c} = (c_x, c_y, c_z).$$

Смешанное произведение будет выглядеть так:

$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix}.$$

➤ Пример на Python

Для нахождения смешанного произведения с использованием *Python* воспользуемся знаниями из раздела про определители матриц:

```
>>> a = [2, 7, 3]
>>> b = [5, 9, 3]
>>> c = [7, 4, 5]
>>> vmp = np.matrix([a, b, c])
>>> np.linalg.det(vmp)
-90.999999999999972
```

Глава 4. Линейные векторные пространства

4.1 Общие сведения

Введем понятие *линейного пространства*. Рассмотрим такое множество X , на котором определена операция сложения элементов из этого множества и умножение элемента множества X на некоторое число. Здесь и далее под числом будем понимать число из множества действительных чисел R . При этом элемент, полученный в результате указанного выше сложения или умножения на число, также принадлежит множеству X .

Если множество X обладает ниже перечисленными свойствами (их называют *аксиомами линейного пространства*), то такое множество называется *линейным пространством*. Если элементами такого линейного пространства являются векторы, то такое пространство называется *линейным векторным пространством*. В рамках данной главы под вектором будем понимать строку или столбце чисел.

Свойство 1. Сложение элементов линейного пространства коммутативно. От перестановки слагаемых сумма не изменяется:

$$a + b = b + a; \quad a, b \in X.$$

➤ Численный пример

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1,2]]).T
>>> b = np.array([[3,4]]).T
>>> L = a + b
>>> R = b + a
>>> print(L)
[[4]
 [6]]
>>> print(R)
[[4]
 [6]]
```


Свойство 2. Сложение элементов линейного пространства ассоциативно. Результат суммирования ряда элементов не зависит от порядка суммирования:

$$a + (b + c) = (a + b) + c; a, b, c \in X.$$

➤ Численный пример

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \left(\begin{pmatrix} 3 \\ 4 \end{pmatrix} + \begin{pmatrix} 5 \\ 6 \end{pmatrix} \right) = \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right) + \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 9 \\ 12 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1,2]]).T
>>> b = np.array([[3,4]]).T
>>> c = np.array([[5,6]]).T
>>> L = a + (b + c)
>>> R = a + b + c
>>> print(L)
[[ 9]
 [12]]
>>> print(R)
[[ 9]
 [12]]
```

Свойство 3. Множество X содержит нулевой элемент, для которого справедливо следующее:

$$a + 0 = a; a \in X.$$

➤ Численный пример

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1,2]]).T
>>> z = np.array([[0,0]]).T
>>> L = a + z
>>> R = a
>>> print(L)
[[1]
 [2]]
>>> print(R)
[[1]
 [2]]
```

Свойство 4. В множестве X для любого элемента существует обратный ему элемент: если a - это элемент множества X , то $(-a)$ - является обратным элементом для a , и он также содержится в множестве X . При этом выполняется равенство:

$$a + (-a) = 0; a \in X.$$

➤ Численный пример

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} -1 \\ -2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1,2]]).T
>>> m_a = np.array([[-1,-2]]).T
>>> z = np.array([[0,0]]).T
>>> L = a + m_a
>>> R = z
>>> print(L)
[[0]
 [0]]
>>> print(R)
[[0]
 [0]]
```

Свойство 5. Произведение суммы элементов из множества X на число равно сумме произведений этого числа на данные элементы:

$$q \cdot (a + b) = q \cdot a + q \cdot b; a, b \in X; q \in R.$$

➤ Численный пример

$$2 \cdot \left(\begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right) = 2 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 8 \\ 12 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1, 2]]).T
>>> b = np.array([[3, 4]]).T
>>> k = 2
>>> L = k * (a + b)
>>> R = k * a + k * b
>>> print(L)
[[ 8]
 [12]]
```

```
>>> print(R)
[[ 8]
 [12]]
```

Свойство 6. Произведение суммы чисел на элемент множества X равно сумме произведений элемента на заданные числа:

$$(q + p) \cdot a = q \cdot a + p \cdot a; a \in X; p, q \in R.$$

➤ Численный пример

$$(2 + 3) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = 2 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} + 3 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 10 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1, 2]]).T
>>> p = 2
>>> q = 3
>>> L = (p + q) * a
>>> R = p * a + q * a
>>> print(L)
[[ 5]
 [10]]
>>> print(R)
[[ 5]
 [10]]
```

Свойство 7. Произведение чисел, умноженное на элемент из множества X , равно произведению одного из чисел и элемента, умноженного на другое число:

$$(q \cdot p) \cdot a = q \cdot (p \cdot a); a \in X; p, q \in R.$$

➤ Численный пример

$$(2 \cdot 3) \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = 2 \cdot \left(3 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right) = \begin{pmatrix} 6 \\ 12 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1, 2]]).T
>>> p = 2
>>> q = 3
>>> L = (p * q) * a
```

```
>>> R = p * (q * a)
>>> print(L)
[[ 6]
 [12]]
>>> print(R)
[[ 6]
 [12]]
```

Свойство 8. Множество X содержит единичный элемент, для которого справедливо следующее равенство:

$$1 \cdot a = a; a \in X.$$

➤ Численный пример

$$1 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

➤ Пример на Python

```
>>> a = np.array([[1, 2]]).T
>>> k = 1
>>> L = k * a
>>> R = a
>>> print(L)
[[1]
 [2]]
>>> print(R)
[[1]
 [2]]
```

4.2 Линейная зависимость векторов

Рассмотрим очень важную тему, которая имеет огромное практическое значение - *линейная зависимость векторов*. Пусть у нас есть система векторов x_1, x_2, \dots, x_n , как вы помните, каждый вектор представляет собой строку или столбец чисел $x_i = (x_{i1}, x_{i2}, \dots, x_{in})$.

Составим следующее равенство:

$$a_1 x_1 + a_2 x_2 + \dots + a_n x_n = 0.$$

Если такое равенство выполняется при a_1, a_2, \dots, a_n не все из которых равны нулю, то такая система векторов x_1, x_2, \dots, x_n называется **линейно зависимой**. Вектор z , который может быть представлен как сумма произведений векторов на числа, называется **линейной комбинацией векторов**:

$$z = a_1 x_1 + a_2 x_2 + \dots + a_n x_n.$$

Для того, чтобы определить является ли заданная система векторов **линейно зависимой**, необходимо разложить вектора на компоненты и составить соответствующую систему уравнений, если она имеет ненулевое решение, то такая система векторов линейно зависима.

➤ Численный пример

Пусть дана следующая система векторов:

$$x_1 = (1, 1, -2),$$

$$x_2 = (-1, 3, 1),$$

$$x_3 = (2, 2, -4).$$

Необходимо определить является ли она линейно зависимой.

Составим систему уравнений:

$$\begin{cases} a_1 - a_2 + 2 \cdot a_3 = 0 \\ a_1 + 3 \cdot a_2 + 2 \cdot a_3 = 0 \\ -2 \cdot a_1 + a_2 - 4 \cdot a_3 = 0 \end{cases}.$$

Полученная система является однородной. Для того, чтобы она имела единственное (нулевое) решение, необходимо, чтобы ее ранг совпадал с числом неизвестных. Ранг главной матрицы системы равен двум, а число неизвестных - три, следовательно система уравнений имеет как минимум одно ненулевое решение, это означает, что система векторов зависима.

В случае, когда число уравнений равно числу неизвестных, для определения числа решений можно ориентироваться на определитель главной матрицы системы. Если он равен нулю, то система имеет бесконечное множество решений, если отличен от

нуля, то она имеет одно (нулевое) решение. Определитель главной матрицы нашей системы равен нулю, что еще раз подтверждает вывод о том, что заданная система векторов линейно зависима.

➤ Пример на Python

Приведем необходимые вычисления на *Python*:

```
>>> A = np.matrix('1 -1 2; 1 3 2; -2 1 -4')
>>> np.linalg.det(A)
0.0
>>> np.linalg.matrix_rank(A)
2
```

4.3 Базис системы векторов

У системы векторов можно выделить **максимальную линейно независимую подсистему**, которая характеризуется тем, что если в нее добавить еще один вектор (из данной системы), то она станет линейно зависимой.

Из приведенного в разделе 4.2 примере, максимальная линейно независимая система векторов выглядит так:

$$x_1 = (1, 1, -2), x_2 = (-1, 3, 1),$$

либо так:

$$x_1 = (2, 2, -4), x_2 = (-1, 3, 1).$$

При этом, добавление к любой из данных подсистем вектора из заданной системы приведет к тому, что полученная система будет линейно зависимой.

Для заданной конечной линейной системы векторов **базисом** называется любая ее максимальная линейно независимая подсистема, а число векторов в базисе называется **рангом** системы векторов. Если вы, по заданным векторам, построите матрицу, то ее ранг будет равен рангу базиса исходной системы векторов. Базис конечной линейной системы - это такой набор векторов, через который может быть выражен любой вектор системы. Другими словами, любой вектор из линейной системы является линейной комбинацией векторов базиса.

Для линейного пространства векторов, так же как и для системы, существует понятие базиса. **Базис линейного пространства** - это упорядоченная независимая система векторов, через которую может быть выражен любой вектор этого пространства. Количество векторов в базисе называется размерностью пространства. Число различных базисов линейного пространства бесконечно. Когда мы говорим, что пространство трехмерно - это значит, что его базис состоит из трех векторов, и любая точка (вектор) пространства может быть определена через вектора базиса.

Чаще всего, на практике, приходится встречаться с **естественным базисом**. Выглядит он так:

$$e_1 = (1, 0, 0, \dots, 0)^T,$$

$$e_2 = (0, 1, 0, \dots, 0)^T,$$

...

$$e_n = (0, 0, 0, \dots, 1)^T.$$

Через естественный базис очень просто выразить любой вектор данного линейного пространства. В этом случае, координаты вектора будут коэффициентами соответствующей линейной комбинации. Рассмотрим вектор:

$$z = (z_1, z_2, \dots, z_n)^T.$$

Построим его линейную комбинацию, выраженную через естественный базис:

$$z = z_1 e_1 + z_2 e_2 + \dots + z_n e_n.$$

Необходимо отметить, что такое представление единственно.

Как было отмечено ранее, количество базисов линейного пространства бесконечно, это значит, что помимо естественного базиса существуют еще и другие. Все базисы линейного пространства связаны между собой. **Мы можем переходить от одного базиса к другому** при необходимости. Разберем этот вопрос более подробно.

Выделим в нашем линейном пространстве два базиса:

$$\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n),$$

$$\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n).$$

Так как каждый вектор пространства может быть выражен через вектора базиса, то выразим каждый вектор \mathbf{b} через вектора \mathbf{a} , получим следующую систему:

$$\begin{cases} b_1 = k_{11}a_1 + k_{21}a_2 + \dots + k_{n1}a_n \\ b_2 = k_{12}a_1 + k_{22}a_2 + \dots + k_{n2}a_n \\ \dots \\ b_n = k_{1n}a_1 + k_{2n}a_2 + \dots + k_{nn}a_n \end{cases}.$$

Тот же результат мы получим если выполним матричное умножение:

$$\mathbf{b} = \mathbf{a} \times K,$$

где \mathbf{a} и \mathbf{b} - это заданные выше вектора, а K - это матрица перехода, которая имеет следующий вид:

$$K = \begin{pmatrix} k_{11} & k_{12} & \dots & k_{1n} \\ k_{21} & k_{22} & \dots & k_{2n} \\ \dots & \dots & \dots & \dots \\ k_{n1} & k_{n2} & \dots & k_{nn} \end{pmatrix}.$$

➤ Численный пример

В параграфе “3.5.5 Векторное произведение двух векторов”, был рассмотрен следующий численный пример:

$$\begin{aligned} \mathbf{c} = \mathbf{a} \times \mathbf{b} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 2 & 7 & 3 \\ 5 & 9 & 3 \end{vmatrix} = \mathbf{i} \cdot \begin{vmatrix} 7 & 3 \\ 9 & 3 \end{vmatrix} - \mathbf{j} \cdot \begin{vmatrix} 2 & 3 \\ 5 & 3 \end{vmatrix} + \mathbf{k} \cdot \begin{vmatrix} 2 & 7 \\ 5 & 9 \end{vmatrix} = \\ &= -6 \cdot \mathbf{i} + 9 \cdot \mathbf{j} - 17 \cdot \mathbf{k}. \end{aligned}$$

Векторы \mathbf{i} , \mathbf{j} , \mathbf{k} , в данном случае являются естественным базисом трехмерного пространства, они имеют координаты:

$$\mathbf{i} = (1, 0, 0),$$

$$\mathbf{j} = (0, 1, 0),$$

$$\mathbf{k} = (0, 0, 1).$$

Эти вектора также называют **орт-векторами**.

4.4 Линейные операторы

Для более интуитивного понимания данной темы, операторы, или как их еще называют отображения, можно воспринимать как функции особого вида. Оператор ставит в соответствие вектору x из линейного пространства X некоторый вектор y из линейного пространства Y . На языке математики это можно записать так:

$$y = f(x), \quad x \in X, \quad y \in Y.$$

Оператор называется линейным, если для него выполняются следующие равенства:

$$f(x_1 + x_2) = f(x_1) + f(x_2) = y_1 + y_2,$$

$$f(\alpha \cdot x_1) = \alpha \cdot f(x_1) = \alpha \cdot y_1,$$

где x_1 и x_2 - это векторы из линейного пространства X ;

y_1, y_2 - векторы из линейного пространства Y ;

α - число из множества действительных чисел R .

Из раздела про базис системы векторов, вы наверное помните, что можно переходить от одного базиса к другому, умножая первый на специальную матрицу перехода.

Для того чтобы задать линейный оператор f , который переводит вектор из линейного пространства X с базисом (a_1, a_2, \dots, a_n) в линейное пространство Y с базисом (b_1, b_2, \dots, b_m) достаточно задать образы $f(a_1), f(a_2), \dots, f(a_n)$, которые переводят базис X в базис Y .

Вектор x , если его представить в виде линейной комбинации векторов базиса, будет выглядеть так:

$$x = x_1 a_1 + x_2 a_2 + \dots + x_n a_n.$$

Тогда отображение вектора x в векторное пространство Y будет представлено следующим образом:

$$f(x) = x_1 f(a_1) + x_2 f(a_2) + \dots + x_n f(a_n).$$

Отображение f векторов базиса линейного пространства X в вектора базиса линейного пространства Y мы уже знаем как строить:

$$\begin{cases} f(\mathbf{a}_1) = c_{11}\mathbf{b}_1 + c_{21}\mathbf{b}_2 + \dots + c_{m1}\mathbf{b}_m \\ f(\mathbf{a}_2) = c_{12}\mathbf{b}_1 + c_{22}\mathbf{b}_2 + \dots + c_{m2}\mathbf{b}_m \\ \dots \\ f(\mathbf{a}_n) = c_{1n}\mathbf{b}_1 + c_{2n}\mathbf{b}_2 + \dots + c_{mn}\mathbf{b}_m \end{cases},$$

$$C_{ba} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix}.$$

Матрица C_{ba} называется **матрицей линейного оператора** f . В отличие от матрицы перехода, которая является квадратной, матрица линейного оператора определяется размерностью пространств, которые связывает данный оператор.

В дальнейшем для нас будут представлять интерес линейные операторы, которые отображают линейное пространство само на себя - случай, когда $X = Y$. Матрицы таких операторов квадратные.

➤ Численный пример

Рассмотрим на примере как действует линейный оператор в линейном векторном пространстве. Пусть нам дано двумерное векторное пространство X и линейный оператор f , действующий в нем. Матрица линейного оператора имеет вид:

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}.$$

Найдем координаты вектора, в который будет переведен с помощью оператора f вектор

$$\mathbf{x} = (1 \ 2)^T,$$

$$\mathbf{x}' = f(\mathbf{x}) = A \cdot \mathbf{x} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \times (1 \ 2)^T = (5 \ 2)^T.$$

.

➤ Пример на Python

На *Python* эта задача решается довольно просто. Для того, чтобы решение было более красивым создадим специальную функцию, которая будет реализовывать данное отображение:

```
>>> f = lambda x: np.dot(np.array([[1, 2], [0, 1]]), x.T)
>>> x = np.array([1, 2])
>>> x_ = f(x)
>>> print(x_)
[5 2]
```

Таким образом получаем, что для того, чтобы выполнить определенные преобразования с вектором необходимо умножить его матрицу линейного оператора. Состав матрицы определяет вид преобразования: отражение, изменение масштаба, поворот и т.п. Рассмотрим на примерах различные виды преобразований.

4.4.1 Отражение вектора в себя

Начнем с самого простого линейного оператора, который переводит вектор в себя.

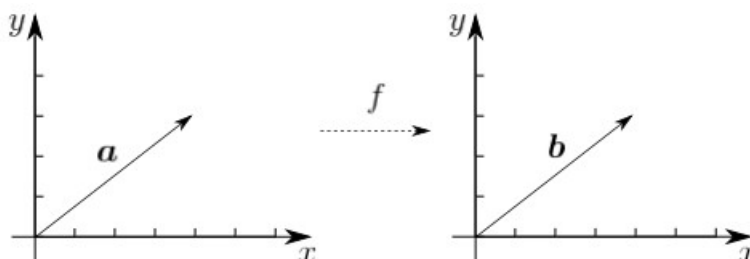


Рисунок 4.1 - Отражение вектора в себя

➤ Численный пример

Здесь и далее будем работать с вектором:

$$x = \begin{pmatrix} 1 & 2 \end{pmatrix}^T.$$

Матрица линейного оператора, отражающая вектор в самого себя, имеет следующий вид:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Вычислим результат применения заданного линейного оператора:

$$x' = f(x) = A \cdot x = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 1 \end{pmatrix}^T = \begin{pmatrix} 2 & 1 \end{pmatrix}^T.$$

➤ Пример на Python

Создадим вектор и матрицу линейного оператора:

```
>>> x = np.array([2, 1])
>>> A = np.array([[1, 0], [0, 1]])
>>> print(A)
[[1 0]
 [0 1]]
```

Создадим функцию f , которая будет реализовывать линейное отражение. Эту функцию будем использовать в следующих примерах:

```
>>> f = lambda A, x: np.dot(A, x.T)
```

Выполним линейное преобразование:

```
>>> f(A, x)
array([2, 1])
```

Как вы можете видеть, результирующий вектор не изменился.

4.4.2 Масштабирование

Матрица, выполняющая масштабирование вектора, имеет следующий вид:

$$A = \begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & k \end{pmatrix}.$$

В таком виде масштаб вектора будет изменен на заданный коэффициент по каждой из осей координат.

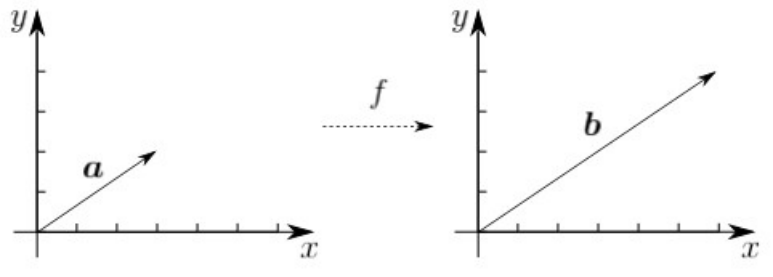


Рисунок 4.2 - Масштабирование вектора

Если необходимо изменить масштаб только в выбранных направлениях, то задайте коэффициенты на нужных позициях главной диагонали, остальным присвойте значение 1. Например, для пространства с базисом (e_1, e_2, e_3) для масштабирования по направлению вектора e_2 можно использовать следующую матрицу:

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

➤ Численный пример

Построим оператор увеличивающий координаты вектора в два раза, его матрица будет иметь вид:

$$A_{m1} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Применим данный оператор к вектору:

$$z = (2 \ 1)^T,$$

$$f(z) = A_{m1} \times z = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \times (2 \ 1)^T = (4 \ 2)^T.$$

Рассмотрим другой пример, по оси абсцисс вектор уменьшим в два раза, а по оси ординат увеличим в три раза:

$$A_{m2} = \begin{pmatrix} 0.5 & 0 \\ 0 & 3 \end{pmatrix}.$$

Тогда результат применения данного вектора к уже известному нам вектору z будет выглядеть так:

$$f(z) = A_{m2} \times z = \begin{pmatrix} 0.5 & 0 \\ 0 & 3 \end{pmatrix} \times (2 \ 1)^T = (1 \ 3)^T.$$

➤ Пример на Python

Выполним вычисления из приведенного выше численного примера:

```
>>> x = np.array([2, 1])
>>> A_m1 = np.array([[2, 0], [0, 2]])
>>> f(A_m1, x)
array([4, 2])
>>> A_m2 = np.array([[0.5, 0], [0, 3]])
>>> f(A_m2, x)
array([1., 3.])
```

4.4.3 Отражение

Отражение относительно той или иной оси является востребованной операцией в прикладных задачах. Рассмотрим **горизонтальное** (относительно оси ординат) и **вертикальное** (относительно оси абсцисс) отражение.

4.4.3.1 Горизонтальное отражение

Оператор, выполняющий горизонтальное отражение, имеет следующую матрицу:

$$A_{hor} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.$$

При умножении на такую матрицу вектор зеркально отражается относительно оси ординат (см. рисунок 4.3).

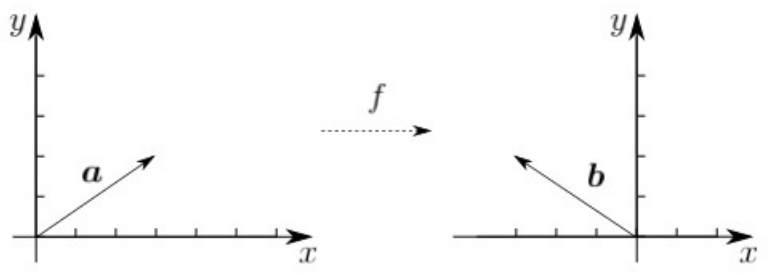


Рисунок 4.3 - Горизонтальное отражение вектора

Рассмотрим численный и программный примеры, демонстрирующие работу данного оператора.

➤ Численный пример

Будем использовать введенную выше матрицу A_{hor} :

$$A_{hor} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.$$

В качестве вектора возьмем уже знакомый нам:

$$z = \begin{pmatrix} 2 & 1 \end{pmatrix}^T.$$

Применим оператор f с матрицей A_{hor} к вектору z :

$$f(z) = A_{hor} \times z = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 1 \end{pmatrix}^T = \begin{pmatrix} -2 & 1 \end{pmatrix}^T.$$

➤ Пример на Python

Решим эту задачу на *Python*:

```
>>> z = np.array([2, 1])
>>> A_hor = np.array([[ -1,  0], [ 0,  1]])
>>> f(A_hor, z)
array([-2,  1])
```

4.4.3.2 Вертикальное отражение

Матрица оператора, выполняющего вертикальное отражение, выглядит следующим образом:

$$A_{vert} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Графически данную операцию можно продемонстрировать так, как показано на рисунке 4.4.

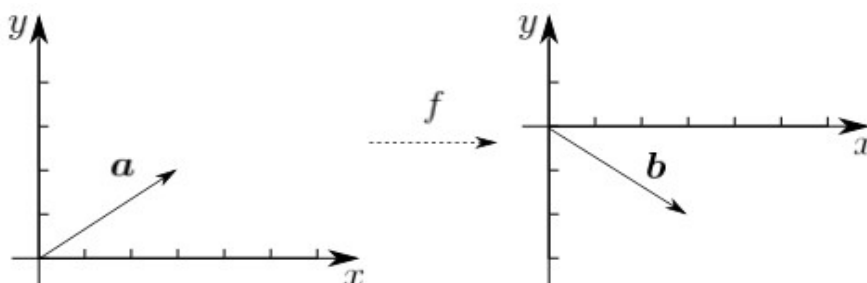


Рисунок 4.4 - Горизонтальное отражение вектора

Перейдем к примерам вертикального отражения.

➤ Численный пример

Воспользуемся заданной матрицей A_{vert} :

$$A_{vert} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Применим оператор f с матрицей A_{vert} к вектору z из предыдущего примера:

$$f(z) = A_{vert} \times z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \times (2 \ 1)^T = (2 \ -1)^T.$$

➤ Пример на Python

Напишем соответствующую программу на *Python*:

```
>>> z = np.array([2, 1])
>>> A_vert = np.array([[1, 0], [0, -1]])
>>> f(A_vert, z)
array([2, -1])
```

4.4.4 Поворот

Матрицы операторов, выполняющих поворот на заданный угол, имеют более интересное наполнение. Рассмотрим два линейных оператора: первый будет осуществлять поворот по часовой стрелке, второй - против.

4.4.4.1 По часовой стрелке

Для поворота по часовой стрелке на угол α используется оператор с матрицей следующего вида:

$$A_{rot_cl} = \begin{pmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{pmatrix}.$$

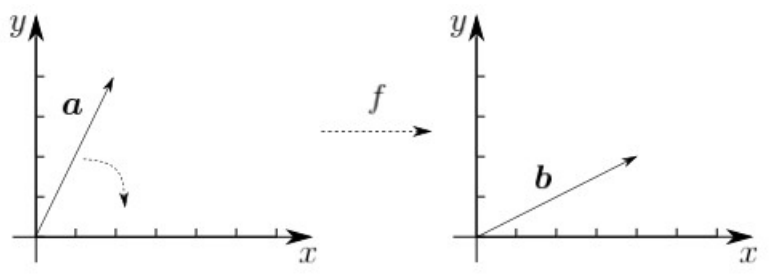


Рисунок 4.5 - Поворот вектора по часовой стрелке

➤ Численный пример

Построим матрицу для оператора, поворачивающего вектор на 45 градусов:

$$A_{rot-cl} = \begin{pmatrix} \cos \frac{\pi}{4} & \sin \frac{\pi}{4} \\ -\sin \frac{\pi}{4} & \cos \frac{\pi}{4} \end{pmatrix}.$$

В качестве аргумента будем использовать вектор:

$$z = \begin{pmatrix} 2 & 1 \end{pmatrix}^T.$$

Применим оператор f с матрицей A_{rot-cl} к вектору z :

$$f(z) = A_{rot-cl} \times z = \begin{pmatrix} \cos \frac{\pi}{4} & \sin \frac{\pi}{4} \\ -\sin \frac{\pi}{4} & \cos \frac{\pi}{4} \end{pmatrix} \times \begin{pmatrix} 2 & 1 \end{pmatrix}^T = \begin{pmatrix} \frac{3\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}^T.$$

➤ Пример на Python

Программа на *Python* с оператором вращения выглядит так:

```
>>> import math
>>> z = np.array([2, 1])
>>> A_rot_cl = np.array([math.cos(math.pi/4), math.sin(math.pi/4)], [(-
1)*math.sin(math.pi/4), math.cos(math.pi/4)])
>>> f(A_rot_cl, z)
array([ 2.12132034, -0.70710678])
```

4.4.4.2 Против часовой стрелке

Для поворота против часовой стрелки на угол α немного модифицируем матрицу из предыдущего параграфа.

$$A_{rot-not-cl} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix}$$

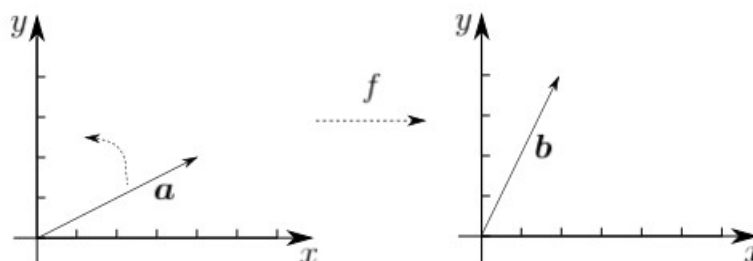


Рисунок 4.6 - Поворот вектора против часовой стрелки

➤ Численный пример

Построим матрицу для оператора, поворачивающего вектор на 60 градусов против часовой стрелки:

$$A_{rot_not_cl} = \begin{pmatrix} \cos \frac{\pi}{3} & -\sin \frac{\pi}{3} \\ \sin \frac{\pi}{3} & \cos \frac{\pi}{3} \end{pmatrix}.$$

В качестве аргумента будет использовать вектор:

$$z = \begin{pmatrix} 2 & 1 \end{pmatrix}^T.$$

Применим оператор f с матрицей $A_{rot_not_cl}$ к вектору z :

$$f(z) = A_{rot_not_cl} \times z = \begin{pmatrix} \cos \frac{\pi}{3} & -\sin \frac{\pi}{3} \\ \sin \frac{\pi}{3} & \cos \frac{\pi}{3} \end{pmatrix} \times \begin{pmatrix} 2 & 1 \end{pmatrix}^T = \begin{pmatrix} 1 - \frac{\sqrt{3}}{2} & \frac{1}{2} + \sqrt{3} \end{pmatrix}^T.$$

➤ Пример на Python

Программа на *Python* с оператором вращения выглядит так:

```
>>> import math
>>> z = np.array([2, 1])
>>> A_rot_not_cl = np.array([[math.cos(math.pi/3), (-1)*math.sin(math.pi/3)],
[math.sin(math.pi/3), math.cos(math.pi/3)]])
>>> f(A_rot_not_cl, z)
array([0.1339746 , 2.23205081])
```

4.4.5 Перемещение по осям

Для решения задачи параллельного переноса векторов в трехмерном пространстве, нам понадобится матрица 3x3, следующего вида:

$$A_{mov} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}.$$

Если мы применим оператор с такой матрицей к вектору:

$$z = \begin{pmatrix} x & y & 1 \end{pmatrix}^T.$$

Получим следующий результат:

$$f(z) = A_{mov} \times z = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ 1 \end{pmatrix}.$$

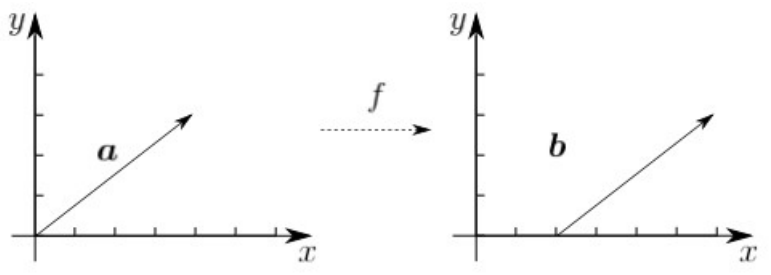


Рисунок 4.7 - Перенос вектора

➤ Численный пример

Матрица, перемещающая вектор на единицу по оси абсцисс и на двойку по оси ординат, будет иметь следующий вид:

$$A_{mov} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Построим вектор, который будет выступать аргументом нашего оператора:

$$z = \begin{pmatrix} 2 & 1 & 1 \end{pmatrix}^T.$$

Применим оператор f с матрицей A_{mov} к вектору z :

$$f(z) = A_{mov} \times z = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \\ 1 \end{pmatrix}.$$

➤ Пример на Python

Программа на *Python* с оператором вращения выглядит так:

```
>>> z = np.array([2,1,1]).T
>>> A_mov = np.array([[1, 0, 1], [0, 1, 2], [0, 0, 1]])
>>> f(A_mov, z)
array([3, 3, 1])
```

4.4.6 Эквивалентные преобразования применительно к СЛАУ

В разделе 2.3 мы разбирали метод Гаусса для решения систем линейных алгебраических уравнений. Наша цель состояла в приведении исходной системы уравнений к треугольному виду. Суть этой работы заключалась в том, что с исходной системой мы последовательно производили ряд преобразований, при этом система до и после этих преобразований имела одно и то же решение. Такие преобразования называются **эквивалентными**.

Различают три вида эквивалентных преобразований:

1. Перестановка двух уравнений в системе.
2. Умножение одного из уравнений системы на число, не равное нулю.
3. Прибавление одного уравнения к другому, умноженному на число.

Применение эквивалентных преобразований удобно производить в матричном виде. Для начала сделаем это вручную.

➤ Численный пример

Вернемся к численному примеру из раздела 2.3: необходимо решить систему линейных уравнений:

$$\begin{cases} 3x - y + 2z = -4 \\ x + 4y - z = 10 \\ 2x + 3y + z = 8 \end{cases}.$$

Построим расширенную матрицу системы:

$$\begin{pmatrix} 3 & -1 & 2 & -4 \\ 1 & 4 & -1 & 10 \\ 2 & 3 & 1 & 8 \end{pmatrix}.$$

Последовательно сделаем с этой системой ряд преобразований, в том же порядке, как это было сделано в разделе 2.3:

$$\begin{pmatrix} 3 & -1 & 2 & -4 \\ 1 & 4 & -1 & 10 \\ 2 & 3 & 1 & 8 \end{pmatrix} \sim \begin{pmatrix} 2 & 3 & 1 & 8 \\ 1 & 4 & -1 & 10 \\ 3 & -1 & 2 & -4 \end{pmatrix} \sim \begin{pmatrix} 2 & 3 & 1 & 8 \\ 0 & -2.5 & -1.5 & -6 \\ 0 & 5.5 & -0.5 & -16 \end{pmatrix} \sim \\ \sim \begin{pmatrix} 2 & 3 & 1 & 8 \\ 0 & -2.5 & -1.5 & -6 \\ 0 & 0 & -2.8 & -2.8 \end{pmatrix}.$$

Каждое из эквивалентных преобразований можно представить как результат применения линейного оператора к заданной матрице.

Рассмотрим матрицы, умножение на которые соответствует определенным эквивалентным преобразованиям.

1. Перестановка двух уравнений в системе.

Данному преобразованию соответствует перестановка строк в расширенной матрице системы. Для того, чтобы произвести перестановку строк достаточно заданную матрицу размера (m, n) умножить на единичную матрицу размера (n, n) , в которой будут переставлены соответствующие строки.

2. Умножение одного из уравнений системы на число, не равное нулю.

Для выполнения такого преобразования необходимо расширенную матрицу системы умножить слева на матрицу, которая используется для масштабирования вектора по заданному направлению:

$$A = \begin{pmatrix} 1 & & & & \\ & \dots & & & 0 \\ & & 1 & & \\ & & & k & \\ & & & & 1 & \\ 0 & & & & & \dots \\ & & & & & & 1 \end{pmatrix}.$$

3. Прибавление одного уравнения к другому, умноженному на число.

Такое эквивалентное преобразование производится путем умножения соответствующей расширенной матрицы системы на матрицу следующего вида:

$$A = \begin{pmatrix} 1 & & & & \\ & \dots & & & 0 \\ & & 1 & & \\ & & & \dots & \\ & & k & & 1 & \\ 0 & & & & & \dots \\ & & & & & & 1 \end{pmatrix}.$$

> Численный пример

Построим и применим соответствующие матрицы для преобразования расширенной матрицы системы из предыдущего примера:

$$A = \begin{pmatrix} 3 & -1 & 2 & -4 \\ 1 & 4 & -1 & 10 \\ 2 & 3 & 1 & 8 \end{pmatrix}.$$

Матрица, выполняющая первое эквивалентное преобразование, меняет 1-ю и 3-ю строки, она имеет следующий вид:

$$R_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Второе эквивалентное преобразование заключается в том, что мы вначале делим первую строку на 2 и вычитаем из нее вторую строку, после этого умножаем первую строку на 3/2 и вычитаем из нее третью строку. Этот набор операций можно произвести с помощью матрицы:

$$R_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & -1 & 0 \\ 1.5 & 0 & -1 \end{pmatrix}.$$

В рамках третьего преобразования мы умножаем вторую строку на $(-11/5)$ и вычитаем из нее третью строку. Построим соответствующую матрицу:

$$R_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2.2 & -1 \end{pmatrix}.$$

Процесс построения треугольной матрицы будет выглядеть так:

$$B = R_3 \times R_2 \times R_1 \times A,$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2.2 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & -1 & 0 \\ 1.5 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 3 & -1 & 2 & -4 \\ 1 & 4 & -1 & 10 \\ 2 & 3 & 1 & 8 \end{pmatrix} = \\ = \begin{pmatrix} 2 & 3 & 1 & 8 \\ 0 & -2.5 & -1.5 & -6 \\ 0 & 0 & -2.8 & -2.8 \end{pmatrix}.$$

➤ Пример на Python

Решим задачу на *Python*. Для начала создадим исходную расширенную матрицу системы:

```
>>> A = np.array([[3, -1, 2, -4], [1, 4, -1, 10], [2, 3, 1, 8]])
>>> print(A)
[[ 3 -1  2 -4]
 [ 1  4 -1 10]
 [ 2  3  1  8]]
```

Теперь построим матрицы для выполнения эквивалентных преобразований:

```
>>> R1 = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
>>> print(R1)
[[0 0 1]
 [0 1 0]
 [1 0 0]]
>>> R2 = np.array([[1, 0, 0], [0.5, -1, 0], [1.5, 0, -1]])
>>> print(R2)
[[ 1.  0.  0. ]
 [ 0.5 -1.  0. ]
 [ 1.5  0. -1. ]]
>>> R3 = np.array([[1, 0, 0], [0, 1, 0], [0, -2.2, -1]])
>>> print(R3)
[[ 1.  0.  0. ]
 [ 0.  1.  0. ]
 [ 0. -2.2 -1. ]]
```

Выполним необходимые матричные умножения:

```
>>> R3.dot(R2.dot(R1.dot(A)))
array([[ 2.,  3.,  1.,  8.],
       [ 0., -2.5,  1.5, -6.],
       [ 0.,  0., -2.8, -2.8]])
```

Как вы можете видеть результат совпадает с тем, что было получено при вычислении вручную.

Если данный вопрос перевести на язык линейных операторов, то можно сделать следующее интересное наблюдение: задав умножение на соответствующую матрицу как линейное отображение, готовый результат мы можем получить через композицию

линейных операторов — то есть последовательное их применение к результату полученному на предыдущем этапе. Рассмотрим это на примере. Создадим три линейных оператора:

```
>>> def f1(x):
    R1 = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
    return R1.dot(x)

>>> def f2(x):
    R2 = np.array([[1, 0, 0], [0.5, -1, 0], [1.5, 0, -1]])
    return R2.dot(x)

>>> def f3(x):
    R3 = np.array([[1, 0, 0], [0, 1, 0], [0, -2.2, -1]])
    return R3.dot(x)
```

Теперь применим их последовательно к заданной матрице:

```
>>> f3(f2(f1(A)))
array([[ 2. ,  3. ,  1. ,  8. ],
       [ 0. , -2.5,  1.5, -6. ],
       [ 0. ,  0. , -2.8, -2.8]])
```

4.5 Собственные векторы линейного оператора

4.5.1 Характеристический полином матрицы

Если из матрицы A вычесть единичную матрицу E , умноженную на некоторую переменную λ , то получим матрицу, которая называется характеристической матрицей матрицы A :

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix},$$

$$A - \lambda E = \begin{pmatrix} a_{11} - \lambda & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} - \lambda & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} - \lambda \end{pmatrix}.$$

Определитель такой матрицы будет полиномом n -ой степени от λ и может рассматриваться как функция:

$$f(\lambda) = |A - \lambda E|.$$

➤ Численный пример

Построим характеристическое уравнение для следующей матрицы:

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{pmatrix},$$

$$\begin{aligned} |A - \lambda E| &= (1 - \lambda) \cdot \begin{vmatrix} 2 - \lambda & 2 \\ 0 & 3 - \lambda \end{vmatrix} - 4 \cdot \begin{vmatrix} 0 & 2 \\ 0 & 3 - \lambda \end{vmatrix} + 1 \cdot \begin{vmatrix} 0 & 2 - \lambda \\ 0 & 0 \end{vmatrix} = \\ &= (1 - \lambda) \cdot (2 - \lambda) \cdot (3 - \lambda). \end{aligned}$$

Решим полученное уравнение:

$$(1 - \lambda) \cdot (2 - \lambda) \cdot (3 - \lambda) = 0,$$

$$\lambda_1 = 1,$$

$$\lambda_2 = 2,$$

$$\lambda_3 = 3.$$

➤ Пример на Python

Воспользуемся возможностями *Python* и библиотеки *Numpy* для вычисления корней характеристического уравнения матрицы:

```
>>> A = np.matrix("1 4 1; 0 2 2 ; 0 0 3")
>>> A
matrix([[1, 4, 1],
        [0, 2, 2],
        [0, 0, 3]])
>>> w, v = np.linalg.eig(A)
>>> w
array([1., 2., 3.]
```

4.5.2 Собственные векторы и собственные значения линейного оператора

В самом начале раздела 4.4 мы ввели понятие линейного оператора. Если линейный оператор переводит некоторый вектор x в вектор λx , то такой вектор называется **собственным вектором** оператора f :

$$f(x) = \lambda x.$$

При этом λx называется **собственным значением** линейного оператора f .

Полученный вектор является коллинеарным, то есть он равен исходному вектору, умноженному на какое-то число.

Собственными значениями линейного оператора могут быть только корни характеристического уравнения матрицы этого линейного оператора. При этом из корней следует выбирать только те, которые принадлежат полю над которым построено линейное пространство. То есть, если у нас элементы вектора - это действительные числа, то из всех собственных значений нужно выбрать те, которые являются действительными числами, а если есть комплексные, что возможно при решении уравнения, то их следует отбросить.

Попробуем самостоятельно построить собственный вектор для заданного оператора.

➤ Численный пример

Для простоты возьмем линейный оператор, работающий с векторами над полем действительных чисел, с матрицей A :

$$A = \begin{pmatrix} 7 & 3 \\ -5 & -1 \end{pmatrix}.$$

Построим характеристическое уравнение матрицы:

$$|A - \lambda E| = \begin{vmatrix} 7 - \lambda & 3 \\ -5 & -1 - \lambda \end{vmatrix} = (7 - \lambda) \cdot (-1 - \lambda) - (-15) = \lambda^2 - 6\lambda + 8.$$

Найдем корни характеристического уравнения:

$$\lambda^2 - 6\lambda + 8 = 0,$$

$$\lambda_{1,2} = \frac{6 \pm \sqrt{36 - 32}}{2} = \frac{6 \pm 2}{2},$$

$$\lambda_1 = 4,$$

$$\lambda_2 = 2.$$

Из корней характеристического уравнения выбираем те, которые являются действительными числами (так как элементы вектора, над которым работает линейный оператор, являются действительными числами). В нашем случае - это все полученные корни.

Построим собственные вектора данного линейного оператора:

$$\begin{pmatrix} 7 & 3 \\ -5 & -1 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = 4 \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

$$\begin{cases} 7 \cdot x_1 + 3 \cdot x_2 = 4 \cdot x_1 \\ -5 \cdot x_1 - 1 \cdot x_2 = 4 \cdot x_2 \end{cases},$$

$$\begin{cases} 3 \cdot x_1 + 3 \cdot x_2 = 0 \\ -5 \cdot x_1 - 5 \cdot x_2 = 0 \end{cases},$$

$$x_2 = -x_1.$$

Для первого корня характеристического уравнения собственный вектор будет иметь следующий вид: $(x_1, -x_1)$.

Найдем собственный вектор для второго корня характеристического уравнения:

$$\begin{pmatrix} 7 & 3 \\ -5 & -1 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = 2 \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

$$\begin{cases} 7 \cdot x_1 + 3 \cdot x_2 = 2 \cdot x_1 \\ -5 \cdot x_1 - 1 \cdot x_2 = 2 \cdot x_2 \end{cases},$$

$$\begin{cases} 5 \cdot x_1 + 3 \cdot x_2 = 0 \\ -5 \cdot x_1 - 3 \cdot x_2 = 0 \end{cases},$$

$$x_2 = -\frac{5}{3} \cdot x_1.$$

Собственный вектор для $\lambda = 2$ имеет вид: $(x_1, -\frac{5}{3} \cdot x_1)$.

Собственные векторы и собственные значения линейного оператора имеют большое практическое значение, особенно в такой области как машинное обучение и анализ данных. Линейный оператор, воздействуя на собственный вектор, как мы только что узнали, изменяет лишь длину этого вектора, направление остается прежним. Другими словами, направление собственного вектора указывает на направление, в котором происходит максимальное изменение вектора, чаще всего это сжатие или растяжение. Представьте себе таблицу с данными, в которой столбцы - это различные признаки, изначально вы не знаете насколько эти признаки информативны, то есть насколько они хорошо описывают представленные данные. Найденные собственные вектора для нашей таблицы-матрицы выявляют те признаки, которые имеют для нас наибольшую ценность, это позволит сжать матрицу, сохранив максимум полезной информации.

Глава 5. Разложения матриц

5.1 LU -разложение матрицы

LU -разложение используется для представления матрицы в виде произведения двух матриц: L - нижней треугольной матрицы и U - верхней треугольной матрицы:

$$A = LU.$$

У матрицы L все элементы главной диагонали равны единице. Матрицу A можно представить в виде LU -разложения, если все главные диагональные миноры не вырождены (отличны от нуля).

Разложение такого типа используется для:

- решения систем линейных уравнений;
- вычисления обратной матрицы;
- вычисления определителя.

Рассмотрим численный пример LU -разложения.

➤ Численный пример

Построим LU -разложение следующей матрицы:

$$A = \begin{pmatrix} 4 & 1 & 3 \\ 8 & 1 & 8 \\ -12 & -4 & -6 \end{pmatrix}.$$

Используя элементарные преобразования приведем ее к верхнему треугольному виду. Более подробно об этом можете прочитать в разделе “4.4.7 Эквивалентные преобразования применительно к СЛАУ”:

$$\begin{aligned} U &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 4 & 1 & 3 \\ 8 & 1 & 8 \\ -12 & -4 & -6 \end{pmatrix} = \\ &= \begin{pmatrix} 4 & 1 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}. \end{aligned}$$

Если произведение матриц, осуществляющих выполнение эквивалентных преобразований, мы выделим в отдельную матрицу и найдем от нее обратную, то это и будет искомая матрица L :

$$R1 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ 3 & 0 & 1 \end{pmatrix},$$

$$R2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

$$T = R1 \times R2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ 3 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ 5 & -1 & 1 \end{pmatrix},$$

$$L = T^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ -3 & -1 & 1 \end{pmatrix}.$$

Проверим вычисления:

$$A = L \times U = \begin{pmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ -3 & -1 & 1 \end{pmatrix} \times \begin{pmatrix} 4 & 1 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 4 & 1 & 3 \\ 8 & 1 & 8 \\ -12 & -4 & -6 \end{pmatrix}.$$

➤ Пример на Python

Проведем вычисления, аналогичные тем, что были проделаны в численном примере, на языке *Python*:

```
>>> A = np.array([[4, 1, 3], [8, 1, 8], [-12, -4, -6]])
```

```
>>> A
```

```
array([[ 4,  1,  3],
       [ 8,  1,  8],
       [-12, -4, -6]])
```

```
>>> R1=np.array([[1, 0, 0],[2, -1, 0],[3, 0, 1]])
```

```
>>> R2=np.array([[1, 0, 0],[0, 1, 0],[0, 1, 1]])
```

```
>>> U = R2.dot(R1.dot(A))
```

```
>>> U
```

```
array([[ 4,  1,  3],
       [ 0,  1, -2],
       [ 0,  0,  1]])
```

```

>>> T = R2.dot(R1)
>>> T
array([[ 1,  0,  0],
       [ 2, -1,  0],
       [ 5, -1,  1]])

>>> L = np.linalg.inv(T)
>>> L
array([[ 1.,  0.,  0.],
       [ 2., -1., -0.],
       [-3., -1.,  1.]])

>>> L.dot(U)
array([[ 4.,  1.,  3.],
       [ 8.,  1.,  8.],
       [-12., -4., -6.]])

```

LU-разложение можно выполнить, используя специальную функцию из библиотеки **scipy**, эту библиотеку необходимо предварительно установить и импортировать. За данный тип разложения отвечает **scipy.linalg.lu**. В результате будут получены матрицы P , L и U . Смысл матриц L и U остается тот же, матрица P - это простая перестановочная матрица.

Рассмотрим как это работает:

```

>>> import scipy.linalg as la
>>> P, L, U = la.lu(A)

>>> P
array([[0., 0., 1.],
       [0., 1., 0.],
       [1., 0., 0.]])

>>> L
array([[1., 0., 0.],
       [-0.66666667, 1., 0.],
       [-0.33333333, 0.2, 1.]])

>>> U
array([[ -12.,  -4.,  -6.],
       [  0., -1.66666667,  4.],
       [  0.,  0.,  .2]])

```

```
>>> A
array([[ 4,  1,  3],
       [ 8,  1,  8],
       [-12, -4, -6]])

>>> P.dot(L.dot(U))
array([[ 4.,  1.,  3.],
       [ 8.,  1.,  8.],
       [-12., -4., -6.]])
```

5.2 QR-разложение матрицы

По определению QR разложение матрицы является представлением матрицы в виде произведения ортогональной матрицы Q и верхней треугольной матрицы R . С понятием верхней треугольной матрицы мы уже встречались ранее. Ортогональная матрица - новое для нас понятие, разберем его более подробно. Ортогональная матрица - это такая матрица, у которой ее транспонированная матрица равна обратной матрице, сумма квадратов всех элементов строки или столбца равна единице, а попарное произведение элементов двух строк или столбцов равно нулю.

Существует несколько способов получения QR разложения матрицы. Рассмотрим один из них, суть которого заключается в ортогонализации и нормировании столбцов исходной матрицы.

➤ Численный пример

Выполним QR разложение для следующей матрицы:

$$A = \begin{pmatrix} 2 & 1 & 3 \\ 0 & 1 & 4 \\ 1 & 3 & 4 \end{pmatrix}.$$

Проведем процедуру ортогонализации над столбцами матрицы A для того, чтобы они стали перпендикулярны друг другу. Выпишем столбцы матрицы:

$$a_1 = (2 \ 0 \ 1)^T,$$

$$a_2 = (1 \ 1 \ 3)^T,$$

$$a_3 = (3 \ 4 \ 4)^T.$$

После процесса ортогонализации получим следующие вектора:

$$b_1 = a_1 = (2 \ 0 \ 1)^T,$$

$$b_2 = a_2 - \frac{(a_2, b_1)}{|b_1|^2} \cdot b_1 = (-1 \ 1 \ 2)^T,$$

$$b_3 = a_3 - \frac{(a_3, b_1)}{|b_1|^2} \cdot b_1 - \frac{(a_3, b_2)}{|b_2|^2} \cdot b_2 = (0.5 \ 2.5 \ -1)^T.$$

Теперь проведем нормировку векторов b_1 , b_2 и b_3 :

$$q_1 = \frac{b_1}{|b_1|} = \frac{1}{\sqrt{5}} \cdot \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix},$$

$$q_2 = \frac{b_2}{|b_2|} = \frac{1}{\sqrt{6}} \cdot \begin{pmatrix} -1 \\ 1 \\ 2 \end{pmatrix},$$

$$q_3 = \frac{b_3}{|b_3|} = \frac{1}{\sqrt{7.5}} \cdot \begin{pmatrix} 0.5 \\ 2.5 \\ -1 \end{pmatrix}.$$

Матрица Q будет иметь следующий вид:

$$Q = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{6}} & \frac{0.5}{\sqrt{7.5}} \\ 0 & \frac{1}{\sqrt{6}} & \frac{2.5}{\sqrt{7.5}} \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{6}} & \frac{-1}{\sqrt{7.5}} \end{pmatrix} \approx \begin{pmatrix} 0.894 & -0.408 & 0.183 \\ 0 & 0.408 & 0.913 \\ 0.447 & 0.816 & -0.365 \end{pmatrix}.$$

В тоже время:

$$q_1 = u_{11}a_1,$$

$$q_2 = u_{12}a_1 + u_{22}a_2,$$

$$q_3 = u_{13}a_1 + u_{23}a_2 + u_{33}a_3.$$

Подставим численные значения в приведенные выше формулы:

$$\frac{1}{\sqrt{5}} \cdot \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \Rightarrow u_{11} = \frac{1}{\sqrt{5}},$$

$$\frac{1}{\sqrt{6}} \cdot \begin{pmatrix} -1 \\ 1 \\ 2 \end{pmatrix} = u_{12} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} + u_{22} \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix} = -\frac{1}{\sqrt{6}} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} + \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix} \Rightarrow$$

$$\Rightarrow u_{12} = -\frac{1}{\sqrt{6}}, u_{22} = \frac{1}{\sqrt{6}},$$

$$\frac{1}{\sqrt{7.5}} \cdot \begin{pmatrix} 0.5 \\ 2.5 \\ -1 \end{pmatrix} = u_{13} \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} + u_{23} \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix} + u_{33} \begin{pmatrix} 3 \\ 4 \\ 4 \end{pmatrix} \Rightarrow$$

$$\Rightarrow u_{13} = -\frac{0.5}{\sqrt{7.5}}, u_{23} = -\frac{1.5}{\sqrt{7.5}}, u_{33} = \frac{1}{\sqrt{7.5}}.$$

Получим матрицу U следующего вида:

$$U = \begin{pmatrix} \frac{1}{\sqrt{5}} & \frac{-1}{\sqrt{6}} & \frac{-0.5}{\sqrt{7.5}} \\ 0 & \frac{1}{\sqrt{6}} & \frac{-1.5}{\sqrt{7.5}} \\ 0 & 0 & \frac{1}{\sqrt{7.5}} \end{pmatrix} \approx \begin{pmatrix} 0.447 & -0.408 & -0.183 \\ 0 & 0.408 & -0.548 \\ 0 & 0 & 0.365 \end{pmatrix}.$$

Так как $Q = AU$, то $A = QU^{-1}$. Это значит, что $A = QR$, где $R = U^{-1}$:

$$R = U^{-1} \approx \begin{pmatrix} 2.236 & 2.236 & 4.472 \\ 0 & 2.449 & 3.674 \\ 0 & 0 & 2.739 \end{pmatrix}.$$

➤ Пример на Python

Рассмотрим быстрый способ получения QR -разложения матрицы, для этого воспользуемся функцией ***np.linalg.qr()*** из библиотеки *Numpy*. В качестве аргумента в эту функцию передается исходная матрица, результатом ее работы являются две матрицы Q и R :

```
>>> A = np.matrix("2 1 3; 0 1 4; 1 3 4")
>>> A
matrix([[2, 1, 3],
        [0, 1, 4],
        [1, 3, 4]])
>>> q, r = np.linalg.qr(A)
```

```
>>> q
matrix([[ -0.89442719,  0.40824829, -0.18257419],
        [          -0., -0.40824829, -0.91287093],
        [ -0.4472136, -0.81649658,  0.36514837]])

>>> r
matrix([[ -2.23606798, -2.23606798, -4.47213595],
        [          0., -2.44948974, -3.67423461],
        [          0.,          0., -2.73861279]])
```

Как вы можете видеть из примера: матрицы Q и R совпадают с теми, что мы получили, в ходе вычисления “вручную” если их умножить на (-1) . Так как произведение двух отрицательных чисел - есть число положительное, то результат можно считать аналогичным.

Если перемножить матрицы q и r , то должны получить матрицу A , проверим это:

```
>>> q.dot(r)
matrix([[2., 1., 3.],
        [0., 1., 4.],
        [1., 3., 4.]])
```

5.3 Сингулярное разложение матрицы

Суть **сингулярного разложения матрицы** заключается в том, ее можно представить в виде произведения трех матриц:

$$A = U\Sigma T^T,$$

где матрица U - ортогональная, Σ - диагональная, T - ортогональная.

Из “Главы 4. Линейные операторы” вы должны помнить, что матрица определяет некоторое линейное преобразование над вектором. Сингулярное разложение матрицы показывает, как такую линейную операцию можно превратить в последовательное вращение вектора, масштабирование (растяжение или сжатие) и снова вращение.

Для сингулярного разложения матрицы мы не будем приводить подробный численный пример, а сразу покажем как эта задача решается на *Python*.

➤ Пример на Python

```
>>> A = np.matrix("1 0 1; 0 1 0; 1 0 1")
```

```
>>> A
```

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

```
>>> u, s, v = np.linalg.svd(A)
```

```
>>> u
```

```
matrix([[ -0.70710678,  0., -0.70710678],
        [  0.,  1.,  0.],
        [ -0.70710678,  0.,  0.70710678]])
```

```
>>> s
```

```
array([2., 1., 0.])
```

```
>>> v
```

```
matrix([[ -0.70710678,  0., -0.70710678],
        [  0.,  1.,  0.],
        [  0.70710678,  0., -0.70710678]])
```

Глава 6. Комплексные числа

6.1 Что такое комплексное число?

Перед тем как перейти к рассмотрению комплексных чисел, вспомним, какие множества чисел используются в математике. Приведенный ниже список не претендует на полноту, он нужен только для того, чтобы плавно перейти к понятию комплексного числа.

В начальной школе начинают проходить, так называемые, **натуральные числа**, они обозначаются буквой \mathbb{N} и используются для счета предметов. Считается, что наименьшим натуральным числом является 1, следующее 2 и т.д.:

$$\mathbb{N} = \{1, 2, \dots, \infty\}.$$

Если мы будем складывать и умножать натуральные числа, то в результате получим также натуральные числа. Но вот для вычитания это не всегда выполнимо, например:

$$5 - 2 = 3,$$

результат - натуральное число.

А в следующем примере:

$$2 - 5 = -3$$

полученное значение уже не является натуральным числом.

Для того, чтобы можно было использовать операцию «вычитание» множество натуральных чисел необходимо расширить до множества целых чисел, оно обозначается \mathbb{Z} , в него входят натуральные числа, обратные им отрицательные числа и ноль:

$$\mathbb{Z} = \{-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty\}.$$

Сложение, вычитание и умножение целых чисел дает целое число. Но для операции «деление» это не так. Рассмотрим пример:

$$4 \div 2 = 2,$$

получаем целое число.

Но если мы делимое и делитель поменяем местами:

$$2 \div 4 = \frac{1}{2},$$

результат уже не будет являться целым числом.

Выполним расширение множества чисел, на этот раз, наше новое множество будет содержать в себе все множество целых чисел и все дроби, которые можно представить в виде отношения числителя и знаменателя, последний из которых не равен нулю. Такое множество называется множество рациональных чисел и обозначается \mathbb{Q} :

$$\frac{a}{b} \in \mathbb{Q}, \text{ если } a \in \mathbb{Z}, b \in \mathbb{Z}, b \neq 0.$$

Помимо простых арифметических действий (сложение, вычитание, умножение и деление) при работе с числами, часто приходится применять операции: возведение в степень и извлечение корня. Опять таки, если взять подходящее рациональное число и извлечь из него квадратный корень:

$$\sqrt{\frac{4}{9}} = \frac{2}{3},$$

получим рациональное число.

По аналогии с изложенным выше, вы уже можете догадаться, что существуют такие рациональные числа, квадратный корень из которых не является рациональным числом, в качестве примера можно привести число 2:

$$\sqrt{2} \approx 1.41421356237$$

Такие числа называются иррациональными. Множества рациональных и иррациональных чисел составляют множество действительных чисел, оно обозначается как \mathbb{R} .

Но операция извлечения квадратного корня достаточно коварна, существуют такие действительные числа, квадратный корень из которых не может быть представлен другим действительным числом. Например квадратный корень из (-1) не является действительным числом. До знакомства с комплексными числами, как правило, даже говорят, что такого корня не существует.

Множество комплексных чисел \mathbb{C} представляет собой множество пар значений (a, b) , где a - называют действительной частью, а b - мнимой частью комплексного числа.

6.2 Задание комплексных чисел

6.2.1 Алгебраическая форма задания комплексного числа

В алгебраической форме комплексное число записывается так:

$$z = a + ib,$$

где a - действительная часть;

b - мнимая часть.

i называют мнимой единицей, квадрат которой равен единице $i^2 = -1$.

Теперь вы знаете, чему равен квадратный корень из (-1) , он равен i :

$$\sqrt{-1} = i.$$

Нулевым элементом в множестве комплексных чисел является следующее:

$$z = 0 + i0.$$

6.2.2 Геометрическая интерпретация комплексного числа

Геометрически комплексное число $z = a + ib$ можно рассматривать как точку на двумерной плоскости, на которой a и b являются координатами. На комплексной плоскости определяют действительную ось, она обозначается Re , и мнимую ось - обозначается Im .

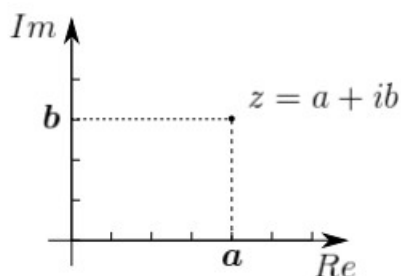


Рисунок 6.1 — Комплексное число на комплексной плоскости

Комплексные числа могут быть представлены в виде векторов, на той же комплексной плоскости, выходящих из точки с координатами $(0, 0)$ до точки (a, b) .

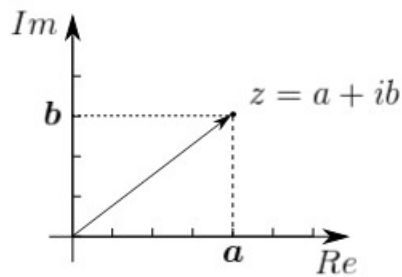


Рисунок 6.2 — Комплексное число, заданное вектором на комплексной плоскости

6.2.3 Модуль и аргумент комплексного числа

Помимо действительной и мнимой части комплексное число может быть определено с помощью модуля и аргумента. Модуль комплексного числа - это длина вектора, а аргумент - угол между вектором и действительной осью.

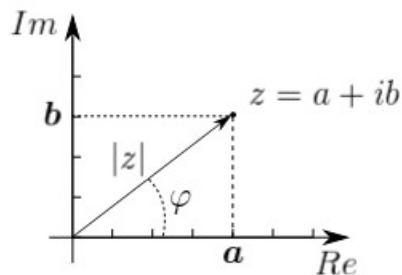


Рисунок 6.3 — Модуль и аргумент комплексного числа

Если комплексное число задано так:

$$z = a + ib,$$

тогда модуль рассчитывается по формуле:

$$|z| = \sqrt{a^2 + b^2},$$

аргумент определяется следующим образом:

$$\varphi = \arg z = \begin{cases} \arctan \frac{b}{a} + \pi \cdot (1 - \operatorname{sign} a), & a \neq 0 \\ \frac{\pi}{2} \cdot \operatorname{sign} b, & a = 0 \end{cases}$$

6.2.4 Тригонометрическая форма задания комплексного числа

Располагая понятиями модуля и аргумента комплексного числа, можно перейти к рассмотрению тригонометрической формы задания комплексного числа.

Если модуль обозначить через r , а аргумент через φ , тогда действительная и мнимая части комплексного числа $z = a + ib$ могут быть вычислены так:

$$a = r \cdot \cos \varphi,$$

$$b = r \cdot \sin \varphi.$$

Само комплексное число примет следующую форму:

$$z = r(\cos \varphi + i \sin \varphi).$$

6.2.5 Показательная форма комплексного числа

Для того, чтобы представить комплексное число в показательной форме необходимо воспользоваться формулой Эйлера:

$$e^{\varphi i} = \cos \varphi + i \sin \varphi.$$

Глядя на тригонометрическую форму записи комплексного числа, можно вывести его показательную форму:

$$z = |z| e^{\varphi i}.$$

6.3 Задание комплексного числа в *Python*

Комплексные числа могут быть заданы парой (a, b) , что сразу наталкивает на мысль использования кортежей *Python* для работы с ними:

```
>>> z = (1, 2)
>>> print(z)
(1, 2)
```

Но в этом случае придется самостоятельно написать функции для выполнения арифметических операций с комплексными числами. Мы этого делать не будем, вместо этого воспользуемся типом *complex*:

```
>>> z = complex(1, 2)
>>> print(z)
(1+2j)
```

```
>>> z1 = complex('1+3j')
>>> print(z1)
(1+3j)
```

Как вы можете видеть, при распечатке такого числа оно представляется в знакомой нам алгебраической форме. Действительная и мнимая части числа могут быть получены с помощью специальных атрибутов:

```
>>> z.real
1.0
>>> z.imag
2.0
```

В типе `complex` нет методов для вычисления модуля и аргумента, для этого можно воспользоваться функциями из модуля `cmath`. Импортируем этот модуль:

```
>>> import cmath
```

Модуль r и аргумент a созданного выше числа z определяется так:

```
>>> r = cmath.polar(z)[0]
>>> print(r)
2.23606797749979
>>> f = cmath.phase(z)
>>> print(f)
1.1071487177940904
```

6.4 Комплексно сопряженное число

Числом, сопряженным с комплексным $z = a + ib$ называют $z = a - ib$. Графически это можно представить так:

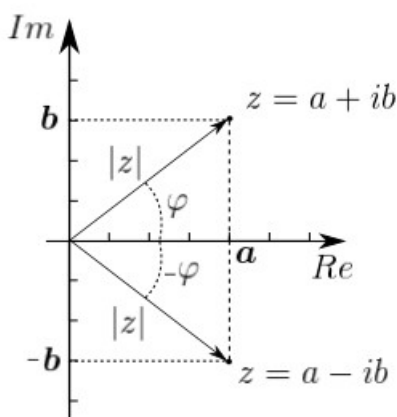


Рисунок 6.4 — Комплексно сопряженное число

➤ Численный пример

Для числа $z = 1 + 2i$ комплексно сопряженным будет $z = 1 - 2i$.

➤ Пример на Python

Создадим комплексное число:

```
>>> z = complex(1, 2)
>>> print(z)
(1+2j)
```

Получим для него комплексно-сопряженное:

```
>>> c_z = z.conjugate()
>>> print(c_z)
(1-2j)
```

6.5 Операции над комплексными числами

6.5.1 Сложение комплексных чисел

При сложении комплексных чисел действительные и мнимые части складываются по отдельности:

$$z = a + ib,$$

$$x = c + id,$$

$$z + x = (a + c) + i(b + d).$$

➤ Численный пример

$$z = 1 + 2i,$$

$$x = 3 + 4i,$$

$$z + x = (1 + 3) + i(2 + 4) = 4 + 6i.$$

➤ Пример на Python

```
>>> z = complex('1+2j')
>>> x = complex('3+4j')
>>> z + x
(4+6j)
```

6.5.2 Вычитание комплексных чисел

При получении разности комплексных чисел вычитание действительных и мнимых частей производится по отдельности:

$$\begin{aligned}z &= a + ib, \\x &= c + id, \\z - x &= (a - c) + i(b - d).\end{aligned}$$

➤ Численный пример

$$\begin{aligned}z &= 5 + 3i, \\x &= 2 + 4i, \\z + x &= (5 - 2) + i(3 - 4) = 3 - 1i.\end{aligned}$$

➤ Пример на Python

```
>>> z = complex('5+3j')
>>> x = complex('2+4j')
>>> z - x
(3-1j)
```

6.5.3 Умножение комплексных чисел

Рассмотрим умножение комплексных чисел. Оно выполняется по алгебраическим правилам умножения сумм в скобках:

$$\begin{aligned}z &= a + ib, \\x &= c + id, \\z \times x &= (a + ib) \times (c + id) = a \times c + i(a \times d) + i(b \times c) + i^2(b \times d) = \\&= (a \times c - b \times d) + i(a \times d + b \times c).\end{aligned}$$

➤ Численный пример

$$\begin{aligned}z &= 5 + 3i, \\x &= 2 + 4i, \\z \times x &= (5 \times 2 - 3 \times 4) + i(3 \times 2 + 5 \times 4) = -2 + 26i.\end{aligned}$$

➤ Пример на Python

```
>>> z = complex('5+3j')
>>> x = complex('2+4j')
>>> z * x
(-2+26j)
```

6.5.4 Деление комплексных чисел

Для вычисления частного от деления двух комплексных чисел используется вот такая формула:

$$\frac{a + ib}{c + id} = \left(\frac{ac + bd}{c^2 + d^2} \right) + i \left(\frac{bc - ad}{c^2 + d^2} \right).$$

➤ Численный пример

$$z = 5 + 3i,$$

$$x = 2 + 4i,$$

$$\frac{z}{x} = \left(\frac{5 \cdot 2 + 3 \cdot 4}{2^2 + 4^2} \right) + i \left(\frac{3 \cdot 2 - 5 \cdot 4}{2^2 + 4^2} \right) = 1.1 - 0.7i.$$

➤ Пример на Python

```
>>> z = complex('5+3j')
>>> x = complex('2+4j')
>>> z / x
(1.1-0.7j)
```

6.6 Возведение в степень комплексного числа

При возведении в степень комплексного числа удобно работать с тригонометрической формой записи числа, это позволит использовать формулу Муавра:

$$[r(\cos \varphi + i \sin \varphi)]^n = r^n (\cos n\varphi + i \sin n\varphi).$$

➤ Численный пример

$$z = 1 + 2i,$$

$$r = \sqrt{1 + 2^2} = \sqrt{5},$$

$$\varphi = \operatorname{atan}\left(\frac{2}{1}\right) = 1.1071487177940904,$$

$$z^2 = (\sqrt{5}(\cos(\varphi) + i \sin(\varphi)))^2 = 5 * (-0.6) + i5 * (0.8) = -3 + i4.$$

➤Пример на Python

```
>>> z = complex('1+2j')
>>> z**2
(-3+4j)
```

6.7 Извлечение корня из комплексного числа

При извлечении корня из комплексного числа, также удобно представлять его в тригонометрической форме. Корень n -ой степени из комплексного числа будет равен n -му количеству корней, вычисляемых по формуле:

$$\sqrt[n]{z} = \sqrt[n]{r} \left(\cos \frac{\varphi + 2k\pi}{n} + i \sin \frac{\varphi + 2k\pi}{n} \right), \quad k = 0, 1, \dots, n-1.$$

➤ Численный пример

$$z = 3 + 4i,$$

$$r = 5,$$

$$\varphi = \operatorname{atan}\left(\frac{4}{3}\right) = 0.927,$$

$$x_0 = \sqrt[5]{5} \left(\cos \frac{\varphi}{2} + i \sin \frac{\varphi}{2} \right) = 2 + i1,$$

$$x_1 = \sqrt[5]{5} \left(\cos \frac{\varphi + 2\pi}{2} + i \sin \frac{\varphi + 2\pi}{2} \right) = -2 - i1.$$

➤Пример на Python

```
>>> import cmath
>>> z = complex('3+4j')
>>> r = abs(z)
>>> ph = cmath.phase(z)
>>> x1 = complex((r**0.5)*math.cos(ph/2), (r**0.5)*math.sin(ph/2))
>>> x1
(2+1j)
>>> x2 = complex((r**0.5)*math.cos((ph+2*cmath.pi)/2),
(r**0.5)*math.sin((ph+2*cmath.pi)/2))
>>> x2
(-2.0000000000000004-0.9999999999999999j)
```

Список литературы

Алгебра / Линейная алгебра

1. Шевцов Г.С. Линейная алгебра: теория и прикладные аспекты: учеб. пособие. - 2-е изд., испр. и доп. - М.: Магистр: ИНФРА-М, 2011. - 528 с.
2. Ильин В.А., Позняк Э.Г. Линейная алгебра: Учеб.: Для вызов. - 6-е изд., стер. - М.: ФИЗМАТЛИТ, 2014.-280 с.
3. Винберг Э.Б. Курс Алгебры. 2-еизд., испр. и доп. — М.: Изд-во «Факториал Пресс», 2001. — 544 с.
4. Кострикин А.И. Введение в алгебру. Часть I. Основы алгебры: Учебник. - М.: Физико-математическая литература, 2000. - 272 с.
5. Кострикин А.И. Введение в алгебру. Часть II. Линейная алгебра. Учебник. - М.: Физико-математическая литература, 2000. — 368 с.
6. Курош А.Г. Курс высшей алгебры. 9-е изд. — М.: Главная редакция физико-математической литературы, 1968. — 431 с.

Python

1. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.,
2. Лучано Рамальо. Python. К вершинам мастерства / Пер. с англ. Слинкин А.А. - М.: ДМК Пресс, 2016. - 768 с.
4. Бизли Д. Python. Подробный справочник. - Пер. с англ. - Спб.: Символ-Плюс, 2010. - 864 с.
5. Alex Martelli. Python in a Nutshell, Second Edition. - O'Reilly Media, Inc.
6. David Beazley, Brian K. Jones. Python Cookbook, Third Edition. - O'Reilly Media, Inc.