Московский государственный университет имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Отчет по заданию №1 «Изучение Python, NumPy» Практикум на ЭВМ 317 группы 2015-2016 учебный год

подготовила студентка 317 учебной группы факультета ВМК МГУ
Шолохова Татьяна

Москва, 9 октября 2015 г.

Содержание

1 Постановка задания

Требовалось для каждой из данных задач:

1. В отдельном Python модуле написать на Python + NumPy не менее трёх вариантов кода различной эффективности(в том числе один полностью векторизованный вариант и один вариант без векторизации).
2. Вычислить скорость работы на нескольких тестовых наборах разного размера.
3. Проанализировать полученные данные о скорости работы разных реализаций.
4. Получить выводы.

2 Проделанная работа

Для каждой из данных задач было сделано:

1. В отдельном Python модуле(task_.py) написано на Python + NumPy три варианта кода различной эффективности(в том числе один полностью векторизованный вариант(v1_vector) и один вариант без векторизации(v2_non_vector)).
2. Сгенерированы тестовые наборы разного размера и вычислена скорость работы на них с помощью IPython Notebook.
3. Проанализированы полученные данные о скорости работы разных реализаций.
4. Получены выводы.
5. Написанный код соответствует style guide PEP 8, что было проверено с помощью утилиты flake8.
6. Ко всем задачам присутствуют автоматические тесты(test_.py), проверяющие совпадение результатов работы всех вариантов кода. Тесты используют встроенный в Python фреймворк unittest.

3 Задача 1

Условие

Подсчитать произведение ненулевых элементов на диагонали прямоугольной матрицы. (Можно считать, что на диагонали матрицы есть ненулевые элементы)

Решение 1. Векторизованное

С помощью функций NumPy сначала находится массив значений на диагонали матрицы($np.diag()$), далее перемножаются($np.prod()$) все ненулевые элементы($np.nonzero()$) этого массива.

```
def v1_vector(X):
    y = np.diag(X)
    return np.prod(y[np.nonzero(y)])
```

Решение 2. Невекторизованное

С помощью цикла for все элементы диагонали матрицы просматриваются, сравниваются с нулем и при необходимости учитываются в ответе.

```
def v2_non_vector(X):
    res = 1
    for i in range(min(X.shape[0], X.shape[1])):
        if X[i, i] != 0:
            res *= X[i, i]
    return res
```

Решение 3

С помощью функций NumPy сначала находится массив значений на диагонали матрицы($np.diag()$), далее с помощью цикла for все значения этого массива просматриваются, сравниваются с нулем и при необходимости учитывются в ответе.

```
def v3_part_vector(X):
    res = 1
    y = np.diag(X)
    for i in y:
        if i != 0:
            res *= i
    return res
```

Вычисление скоростей

В модуле $task1.py$ описана функция $gen(size)$, которая в зависимости от значения параметра $size$ генерирует случайные тестовые данные разных объёмов. В данном случае матрицу X со значениями: -1, 0, 1.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение $size$	0	1	2	3	4
Размеры матрицы X	50×30	100×120	520×500	1000×1020	5020×5000
Время работы векторного решения	38 μs	40 μs	59 μs	159 μs	0.73 ms
Время работы невекторного решения	59 μs	193 μs	1 ms	1.8 ms	10 ms
Время работы решения 3	51 μs	119 μs	0.5 ms	1 ms	5 ms

Вывод

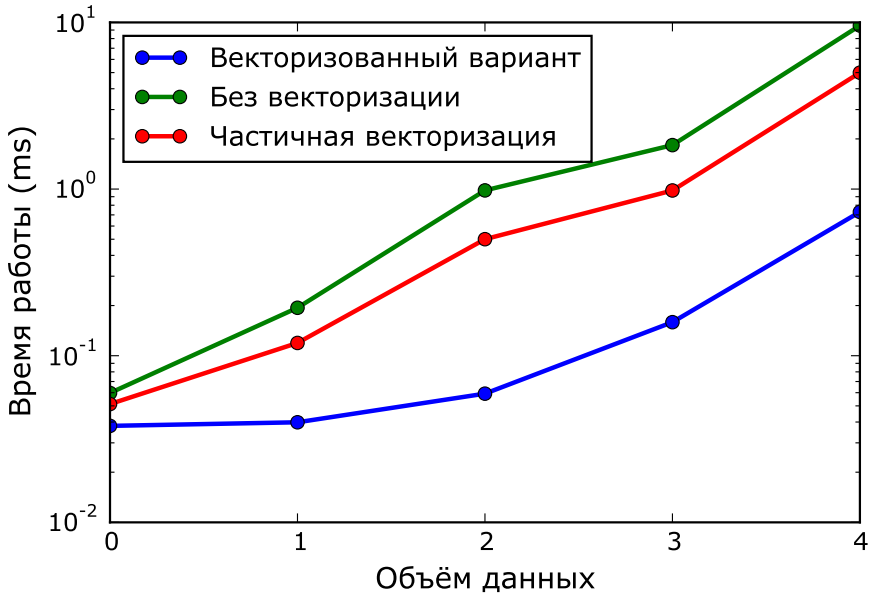

Рис. 1: Задача 1. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 1, векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Это соответствует теоретическим соображениям.

4 Задача 2

Условие

Дана матрица X и два вектора одинаковой длины i и j.
Построить вектор $np.array([X[i[0], j[0]], X[i[1], j[1]], \ldots, X[i[N-1], j[N-1]]])$

Решение 1. Векторизованное

Используется сложная индексация NumPy

```
def v1_vector(X, i, j):
    return X[i, j]
```

Решение 2. Невекторизованное

С помощью цикла for просматриваются все значения векторов i, j, и нужный элемент матрицы X добавляется в ответ.

```
def v2_non_vector(X, i, j):
    res = []
    for k in range(0, i.shape[0]):
        res.append(X[i[k], j[k]])
    return res
```

Решение 3

Дважды используется сложная индексация NumPy.

```
def v3_part_vector(X, i, j):
    x = X[i]
    return x[np.arange(0, j.shape[0]), j]
```

Вычисление скоростей

В модуле *task2.py* описана функция $gen(size)$, которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
Размеры матрицы X	70×50	100×120	520×500	1000×1020	5020×5000
Размеры векторов i, j	60	110	510	1010	5010
Время работы векторного решения	18 μs	21 μs	40 μs	73 μs	0.44 ms
Время работы невекторного решения	134 μs	227 μs	1.1 ms	2.2 ms	12.1 ms
Время работы решения 3	48 μs	72 μs	0.45 ms	5 ms	125 ms

Вывод

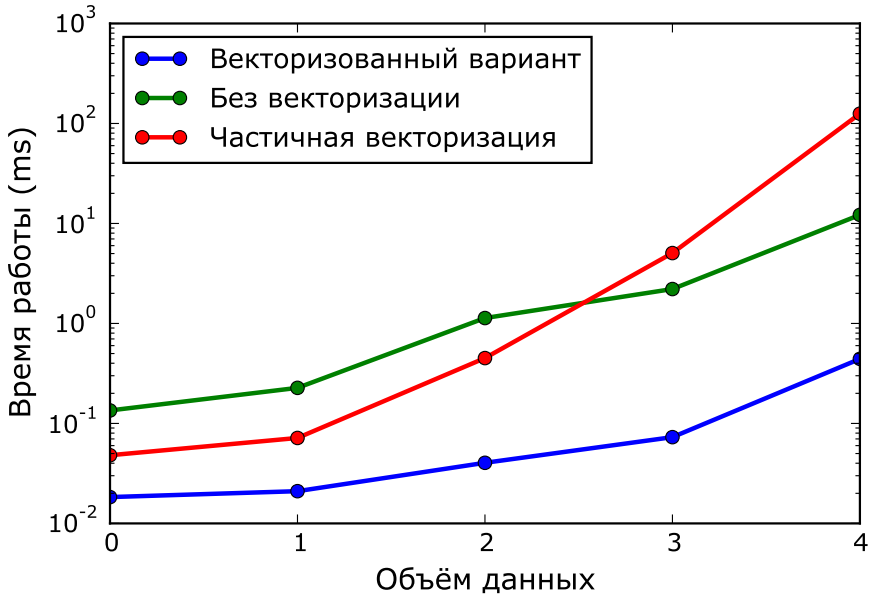

Рис. 2: Задача 2. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 2, векторная реализация работает быстрее остальных. Частично-векторная реализация работает быстрее невекторной на маленьких объемах данных. Но при росте объема данных замедляется, так как при выборе всех нужных строк сохраняется много лишней информации.

5 Задача 3

Условие

Даны два вектора x и y. Проверить, задают ли они одно и то же мультимножество.

Решение 1. Векторизованное

Чтобы проверить два массива на совпадение в смысле мультимнжеств, можно оба массива отсортировать и сравнить соответствующие элементы. Используется сортировка из NumPy($np.sort()$) и функция ($np.all()$) для проверки совпадения всех элементов.

```
def v1_vector(x, y):
    return np.all(np.sort(x) == np.sort(y))
```

Решение 2. Невекторизованное

Чтобы проверить два массива на совпадение в смысле мультимнжеств, можно оба массива отсортировать и сравнить как $list$. Используется стандартная функция($sorted$) языка Python.

```
def v2_non_vector(x, y):
    return sorted(x) == sorted(y)
```

Решение 3

Чтобы проверить два массива на совпадение в смысле мультимнжеств, можно оба массива отсортировать и сравнить соответствующие элементы. Используется сортировка из NumPy($np.sort()$) и с помощью цикла for проверяется совпрадение всех значений.

```
def v3_part_vector(x, y):
    z = np.sort(x) == np.sort(y)
    res = True
    for i in z:
        res = res and i
    return res
```

Вычисление скоростей

В модуле *task3.py* описана функция $gen(size)$, которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Времена измеряются в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
Размеры векторов x, y	10	100	1000	10000	100000
Время работы векторного решения	78 μs	96 μs	162 μs	1.2 ms	12.7 ms
Время работы невекторного решения	30 μs	0.3 ms	2.6 ms	18 ms	180 ms
Время работы решения 3	65 μs	118 μs	288 μs	2.8 ms	28.4 ms

Вывод

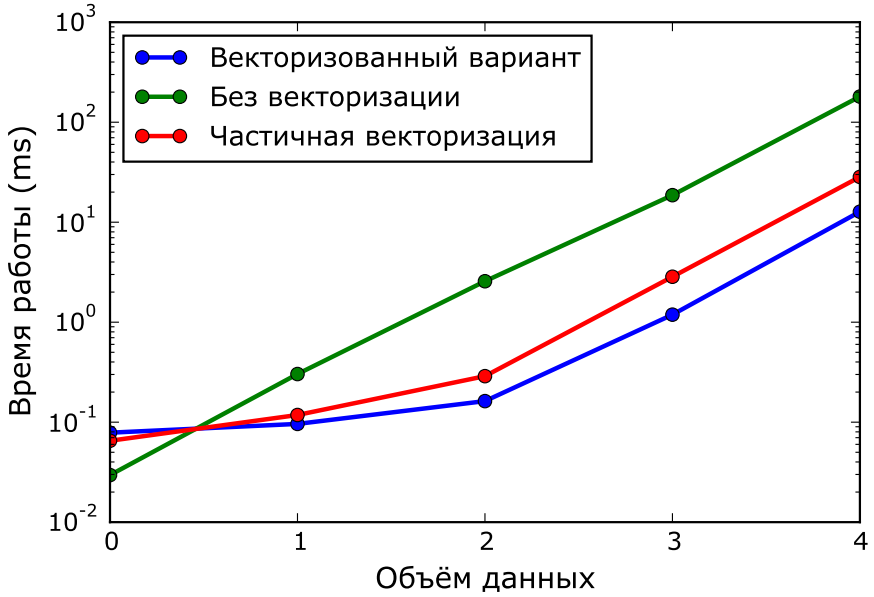

Рис. 3: Задача 3. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 3, на больших объёмах данных, векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Но на маленьких объёмах очерёдность другая, так как время работы всех реализаций очень мало и сравнимо с погрешностью вычисления этого времени.

6 Задача 4

Условие

Найти максимальный элемент в векторе x среди элементов, перед которыми стоит нулевой.

Решение 1. Векторизованное

Используются функции NumPy: $np.max()$ и $np.where()$.

```
def v1_vector(x):
    return np.max(x[np.where(x[0:-1] == 0)[0] + 1])
```

Решение 2. Невекторизованное

С помощью цикла for каждый элемент вектора x проверяется.

```
def v2_non_vector(x):
    res = -np.inf
    for i in range(1, len(x)):
        if (x[i - 1] == 0) and (x[i] > res):
            res = x[i]
    return res
```

Решение 3

С помощью функции $np.where()$ выбираются элементы, перед которыми в векторе 0. Далее в цикле по выбранным элементам находится максимум.

```
def v3_part_vector(x):
    x = x[np.where(x[0:-1] == 0)[0] + 1]
    res = -np.inf
    for i in x:
        if res < i:
            res = i
    return res
```

Вычисление скоростей

В модуле *task4.py* описана функция *gen*(*size*), которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
Размеры вектора x	10	100	1000	10000	100000
Время работы векторного решения	47.6 μs	48 μs	64 μs	271 μs	2.2 ms
Время работы невекторного решения	32 μs	150 μs	1.4 ms	14 ms	134 ms
Время работы решения 3	35 μs	51 μs	178 μs	1.8 ms	15 ms

Вывод

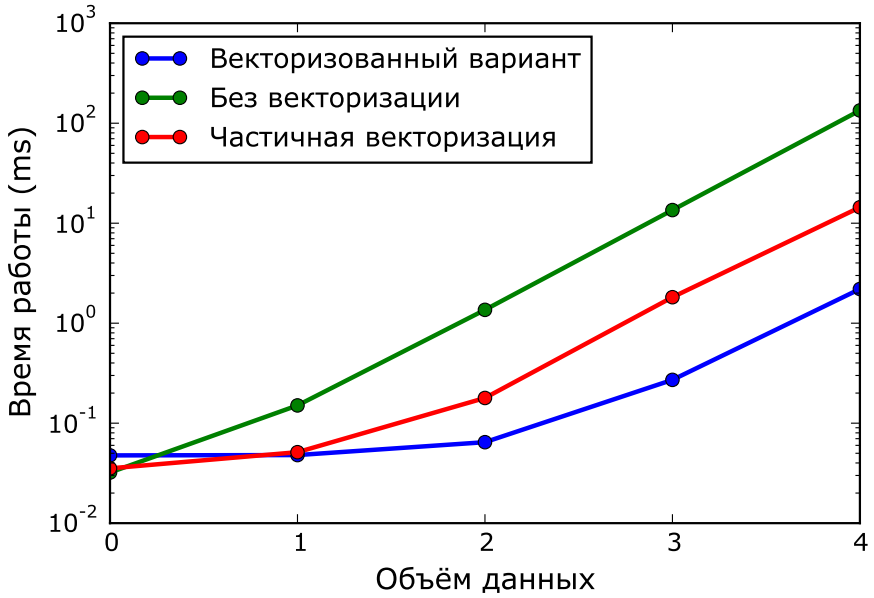

Рис. 4: Задача 4. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 4, на больших объёмах данных векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Но на маленьких объёмах очерёдность другая, так как время работы всех реализаций очень мало и сравнимо с погрешностью вычисления этого времени.

7 Задача 5

Условие

Дан трёхмерный массив, содержащий изображение, размера $(height, width, numChannels)$, а также вектор длины $numChannels$. Сложить каналы изображения с указанными весами, и вернуть результат в виде матрицы размера $(height, width)$. Преобразуйте цветное изображение в оттенки серого, использовав коэффициенты $np.array([0.299, 0.587, 0.114])$.

Решение 1. Векторизованное

Используется broadcasting.

```
def v1_vector(img, ch):
    return np.sum(img * ch[np.newaxis, np.newaxis, :], axis=2)
```

Решение 2. Невекторизованное

Все элементы трёхмерного массива img просматриваются и добавляются в результат с соответствующими коэффициентами вектора ch.

```
def v2_non_vector(img, ch):
    h = img.shape[0]
    w = img.shape[1]
    res = np.zeros((h, w))
    for i in range(0, h):
        for j in range(0, w):
            for c in range(0, ch.shape[0]):
                res[i, j] += img[i, j, c] * ch[c]
    return res
```

Решение 3

Для каждого элемента вектора каналов с помощью broadcasting вычисляется двумерная матрица, которая добавляется в результирующую матрицу.

```
def v3_part_vector(img, ch):
    h = img.shape[0]
    w = img.shape[1]
    res = np.zeros((h, w))
    for c in range(0, ch.shape[0]):
        res += img[:, :, c] * ch[c]
    return res
```

Вычисление скоростей

В модуле *task*5.*py* описана функция *gen*(*size*), которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
height	10	20	40	80	160
width	15	30	60	120	240
numChannels	3	5	10	20	50
Время работы векторного решения	75 μs	94 μs	269 μs	2.3 ms	21 ms
Время работы невекторного решения	6 ms	39 ms	419 ms	3.3 s	32 s
Время работы решения 3	123 μs	188 μs	0.6 ms	3.3 ms	37 ms

Рис. 5: Задача 5. Пример преобразование в оттенки серого

Вывод

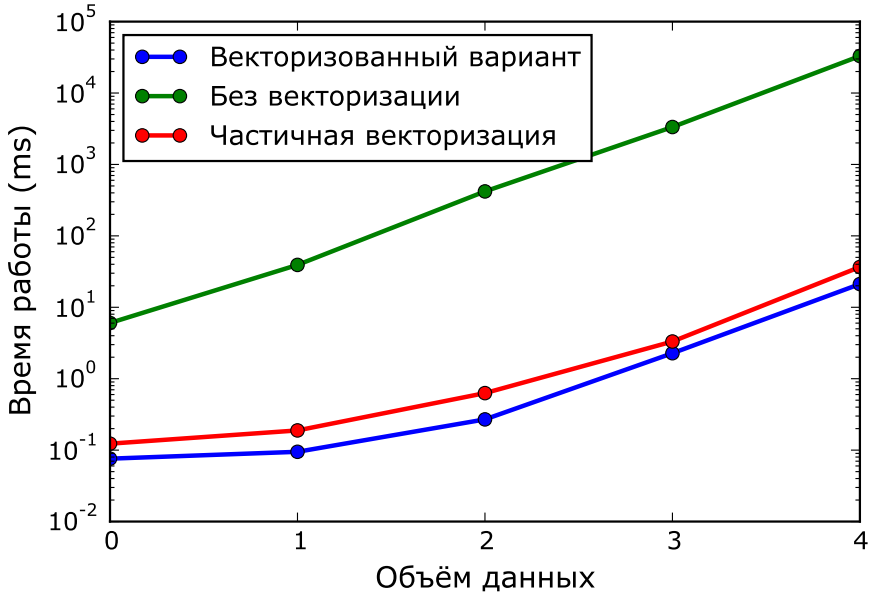

Рис. 6: Задача 5. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 6, векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Это соответствует теоретическим соображениям.

8 Задача 6

Условие

Реализовать кодирование длин серий (Run-length encoding). Дан вектор x. Необходимо вернуть кортеж из двух векторов одинаковой длины. Первый содержит числа, а второй — сколько раз их нужно повторить.

Решение 1. Векторизованное

$jump$ — вектор таких индексов i вектора x, для которых $x[i] - x[i-1] \neq 0$, включая 0 и $len(x)$. Во 2 строке, с помощью функций NumPy вычисляется вектор $jump$. Ответ на задачу вычисляется с помощью сложной индексации $NumPy$.

```
1  def v1_vector(x):
2      jump = np.concatenate(([0], np.where(np.diff(x) != 0)[0] + 1, [len(x)]))
3      return (x[jump[0:len(jump) - 1]], np.diff(jump))
```

Решение 2. Невекторизованное

Просматриваются все значения вектора x и учитываются в ответе.

```
1  def v2_non_vector(x):
2      values = [x[0]]
3      repeats = [1]
4      for i in range(1, len(x)):
5          if x[i] == values[-1]:
6              repeats[-1] += 1
7          else:
8              values.append(x[i])
9              repeats.append(1)
10     return (values, repeats)
```

Решение 3

$jump$ — вектор таких индексов i вектора x, для которых $x[i] - x[i-1] \neq 0$, включая 0. Вектор $jump$ вычисляется без использования функций $NumPy$, а ответ с помощью сложной индексации $NumPy$.

```
1  def v3_part_vector(x):
2      jump = [0]
3      for i in range(1, len(x)):
4          if x[i] != x[i-1]:
5              jump.append(i)
6      return (x[jump], np.diff(jump + [len(x)]))
```

Вычисление скоростей

В модуле *task6.py* описана функция *gen*(*size*), которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
Размер вектора x	100	1000	10000	100000	1000000
Время работы векторного решения	123 μs	209 μs	1.0 ms	1.2 ms	1.4 ms
Время работы невекторного решения	0.7 ms	7.2 ms	73 ms	714 ms	7.6 s
Время работы решения 3	0.7 ms	6.8 ms	67.7 ms	670 ms	7.1 s

Вывод

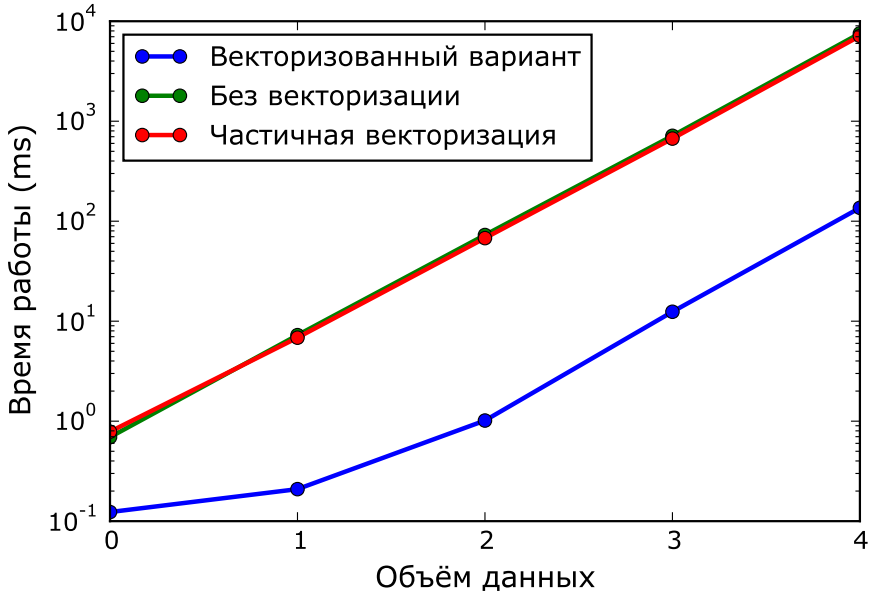

Рис. 7: Задача 6. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 7, векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Это соответствует теоретическим соображениям.

9 Задача 7

Условие

Даны две выборки объектов — X и Y. Вычислить матрицу евклидовых расстояний между объектами.

Решение 1. Векторизованное

С помощью list comprehension для каждого вектора из X вычисляется расстояние до всех векторов Y(с помощью broadcasting).

```
def v1_vector(X, Y):
    return np.array([np.sqrt(np.sum((Y - X[i])**2, axis=1))
                     for i in range(0, X.shape[0])])
```

Решение 2. Менее векторизованное

С помощью цикла for для всех пар векторов $(x, y) : x \in X, y \in Y$ функциями NumPy вычисляется расстояние.

```
def v2_non_vector(X, Y):
    distances = np.zeros((X.shape[0], Y.shape[0]))
    for i in range(0, X.shape[0]):
        for j in range(0, Y.shape[0]):
            distances[i, j] = np.sqrt(np.sum((X[i] - Y[j])**2))
    return distances
```

Решение 3. С использованием SciPy

```
def v3_part_vector(X, Y):
    return scipy.spatial.distance.cdist(X, Y)
```

Вычисление скоростей

В модуле $task7.py$ описана функция $gen(size)$, которая в зависимости от значения параметра $size$ генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение $size$	0	1	2	3	4
Размеры матрицы X	1×13	10×8	100×5	1000×3	10000×2
Размеры матрицы Y	10×13	10×8	10×5	10×3	10×2
Время работы векторного решения	90 μs	0.59 ms	5.4 ms	54 ms	544 ms
Время работы решения 2	457 μs	4.4 ms	42 ms	445 ms	4.4 s
Время работы решения SciPy	106 μs	108 μs	147 μs	574 μs	3.9 ms

Вывод

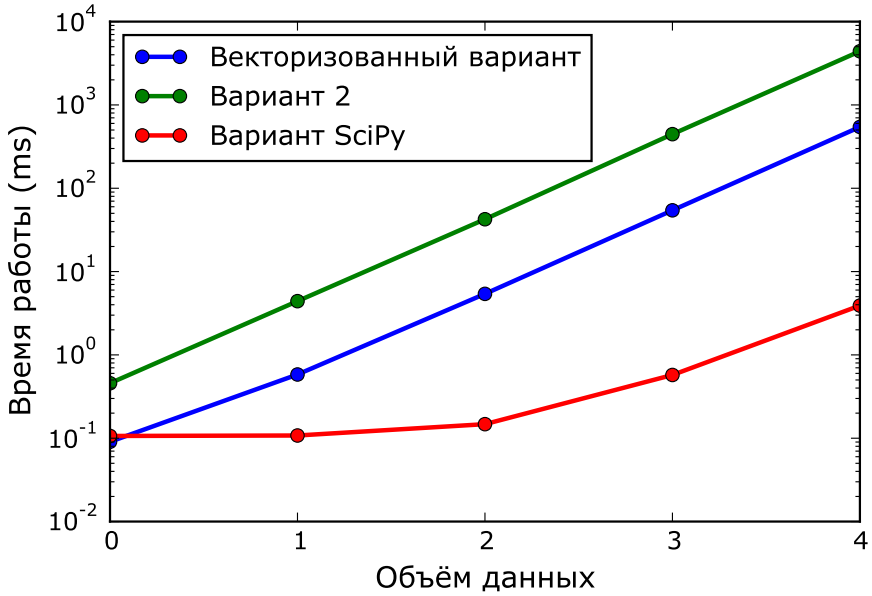

Рис. 8: Задача 7. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 8, реализация SciPy работает быстрее остальных, а векторная лучше менее-векторной. Это соответствует теоретическим соображениям.

10 Задача 8

Условие

Реализовать функцию вычисления логарифма плотности многомерного нормального распределения
Входные параметры: точки X, размер (N, D), мат. ожидание m, вектор длины D, матрица ковариаций C, размер (D, D). Сравнить с $scipy.stats.multivariate_normal(m, C).logpdf(X)$ как по скорости работы, так и по точности вычислений.

Решение 1. Векторизованное

Формула плотности невырожденного нормального распределения выполнена для всей матрицы X.

```
def v1_vector(X, m, C):
    n = m.shape[0]
    ans = -(n/2.0)*np.log(2*np.pi) - 0.5*np.linalg.slogdet(C)[1]
    ans -= 0.5*np.dot(np.dot((X-m), np.linalg.inv(C)), (X-m).T)
    return np.diag(ans)
```

Решение 2. Менее векторизованное

Формула плотности невырожденного нормального распределения выполнена для каждого вектора матрицы X.

```
def v2_non_vector(X, m, C):
    ans = []
    n = m.shape[0]
    for i in range(X.shape[0]):
        x = X[i]
        ans.append(-(n/2.0)*np.log(2*np.pi) - 0.5*np.linalg.slogdet(C)[1] -
                   0.5*np.dot(np.dot((x-m), np.linalg.inv(C)), (x-m).T))
    return np.array(ans)
```

Решение 3. С использованием SciPy

```
def v3_part_vector(X, m, C):
    return scipy.stats.multivariate_normal(m, C).logpdf(X)
```

Вычисление скоростей

В модуле *task8.py* описана функция *gen*(*size*), которая в зависимости от значения параметра *size* генерирует случайные тестовые данные разных объёмов.
Время измеряется в IPython Notebook функцией %timeit -o -q.

Значение *size*	0	1	2	3	4
Размеры матрицы X	5×5	10×10	20×20	40×40	80×80
Время работы векторного решения	315 μs	370 μs	0.5 ms	0.8 ms	1.8 ms
Время работы решения 2	1.3 ms	2.8 ms	7 ms	19.8 ms	103 ms
Время работы SciPy решения	1 ms	1.2 ms	1.6 ms	2.4 ms	5.5 ms

Вывод

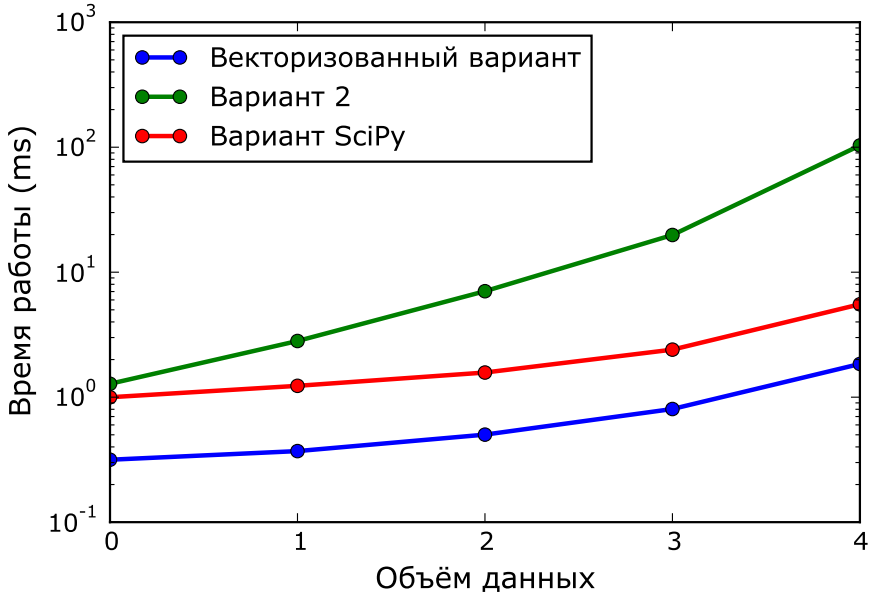

Рис. 9: Задача 8. График зависимости скоростей разных реализаций от объёма входных данных

Как видно на рисунке 9, векторная реализация работает быстрее остальных, частично-векторная реализация работает быстрее невекторной, но медленнее векторной, и невекторизованная работает медленнее остальных. Это соответствует теоретическим соображениям.

Опытным путём выяснено, что абсолютная погрешность прямопропорционально зависит от объёма исходных данных и колеблется от 0 до 10^{-12}

11 Вывод

По проделанной работе можно заключить, что векторный вариант NumPy в сравнении с невекторным работает на несколько порядков быстрее. Использование библиотеки NumPy значительно ускоряет написание программы и работу кода, иногда в ущерб удобству прочтения. Но векторизованный вариант написания программы не всегда оправдан, например время для поготовки маленьких объёмов данных для NumPy сопоставимо со временем работы невекторного кода.