Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет

По заданию №1 по практикуму на ЭВМ

Выполнил студент 317 группы
Измаилов Павел Алексеевич

Москва, 9 октября 2015

1 Описание проделанной работы

В данном отчете содержатся результаты экспериментов, проведенных мной в соответствии с первым заданием по курсу практикума на ЭВМ на кафедре ММП ВМК МГУ. Задание не содержало творческой или теоретической части. Все необходимые эксперименты были проведены, и по ним были получены соответствующие выводы.

Для каждой задачи мной было реализовано три варианта решения — полностью векторизованный, полностью не векторизованный и написанный в стиле функционального программирования.

Весь код был написан мной в соответствии с набором правил PEP 8. Для каждой задачи были написаны автоматические тесты, проверяющие совпадение результатов работы всех вариантов кода.

2 Эксперименты

В данном разделе описываются задачи, рассматриваемые в работе, их решения, проведенные эксперименты и полученные результаты.

Для каждого решения среднее время работы оценивалось с помощью команды `%timeit -n 1000`, кроме случаев, когда это оговорено отдельно.

2.1 Задача 1

В первой задаче требовалось вычислить произведение ненулевых элементов, стоящих на диагонали прямоугольной матрицы $M \in \mathbb{R}^{n \times m}$.

В невекторизованное решении данной задачи произведение ненулевых диагональных элементов вычисляется в цикле по всем диагональным элементам. В векторизованном решении для выделения ненулевых диагональных элементов используется *слайсинг*, после чего они перемножаются с помощью функции `numpy.prod()`. В функциональном решении с помощью *list comprehension* строится список `lst` всех диагональных элементов, и его ненулевые элементы перемножаются с помощью функции `reduce((lambda a, b: a*b + a*int(a*b == 0), lst, 1)`.

Для данной задачи было проведено три эксперимента для случайных матриц M разных размеров, у которых четверть диагональных элементов равна 0. Результаты экспериментов приведены в таблице 1.

Параметры задачи		Время работы		
n	m	Невект. реализация	Вект. реализация	Функ. реализация
100	30	18 μs	13.6 μs	37.9 μs
1000	1000	523 μs	32 μs	1.12 ms
5000	7000	2.45 ms	128 μs	5.89 ms

Таблица 1: Результаты экспериментов по первой задаче

2.2 Задача 2

Во второй задаче требовалось построить вектор из элементов матрицы $M \in \mathbb{R}^{n \times m}$, координаты которого — соответствующие элементы двух векторов равной длины $i, j \in \mathbb{R}^l$.

Невекторизованное решение в цикле строит список всех требуемых элементов матрицы. Векторизованное решение строит необходимую подматрицу с помощью *слайсинга*. Функциональное решение строит список требуемых элементов с помощью *list comprehension*.

Параметры задачи			Время работы		
n	m	l	Невект. реализация	Вект. реализация	Функ. реализация
100	30	15	14.7 μs	2.41 μs	14.9 μs
1000	1000	700	403 μs	9.4 μs	319 μs
5000	7000	2000	1.14 ms	41.8 μs	900 μs

Таблица 2: Результаты экспериментов по второй задаче

Для этой задачи было проведено три эксперимента для случайных матриц M разных размеров и векторов i и j с различными длинами l. Результаты экспериментов приведены в таблице 2.

2.3 Задача 3

В этой задаче требовалось проверить, совпадают ли мультимножества `mul(i)` и `mul(j)`, заданные данными векторами $i, j \in \mathbb{R}^l$.

Все решения основываются на сортировке данных векторов. Невекторизованное решение сортирует списки из элементов векторов с помощью функции `sorted`, после чего последовательно сравнивает получившиеся списки в цикле. Векторизованное решение использует функцию сортировки векторов `numpy.sort()`, после чего сравнивает отсортированные вектора с помощью `numpy.all()`. Функциональное решение сравнивает отсортированные с помощью `sorted()` списки элементов векторов с помощью функции `reduce()`.

Для этой задачи было проведено шесть экспериментов: по три эксперимента с разными l для векторов, задающих одинаковое и разное мультимножество. В случае одинакового мультимножества генерировался случайный вектор v заданной длины, после чего вектора i и j получались случайным перетасовыванием его элементов. В случае, когда генерировались разные мультимножества, вектора i и j были взяты случайными. В таблице 3 приведены результаты работы алгоритмов.

2.4 Задача 4

В этой задаче требовалось найти максимальный элемент в векторе $x \in \mathbb{R}^l$, среди элементов, перед которыми в x стоит 0.

В невекторизованном решении элементы вектора последовательно просматриваются в цикле, и среди элементов, стоящие после 0 выбирается максимум. Векторизованное решение приведено в листинге 1. В функциональном решении список элементов, стоящих в векторе после нулей, строится с помощью *list comprehension*, после чего в нем ищется максимум с помощью функции `max()`.

Листинг 1: Векторизованное решение задачи 4

Параметры задачи		Время работы		
l	`mul(i) == mul(j)`	Невект. реализация	Вект. реализация	Функ. реализация
30	True	20.1 μs	17 μs	31.2 μs
30	False	18.7 μs	16.5 μs	31.2 μs
1000	True	788 μs	87.6 μs	842 μs
1000	False	707 μs	86.2 μs	825 μs
10000	True	9.36 ms	1.08 ms	10.6 ms
10000	False	8.56 ms	1.11 ms	11 ms

Таблица 3: Результаты экспериментов по третьей задаче

```
def vec_maximum_in_front_of_zero(vec):
    return np.max((np.roll(vec, -1))[vec[:-1] == 0])
```

Для этой задачи было проведено три эксперемента для различных l. Для каждого эксперемента был сгенерирован случайный вектор x, после чего в него было добавлено некоторое количество нулей. Результаты экспериментов приведены в таблице 4.

Параметры задачи	Время работы		
l	Невект. реализация	Вект. реализация	Функ. реализация
30	9.22 μs	20.5 μs	14.2 μs
1000	222 μs	27.9 μs	295 μs
10000	2.3 ms	99 μs	2.94 ms

Таблица 4: Результаты экспериментов по четвертой задаче

2.5 Задача 5

В данной задаче требовалось написать функцию, складывающую каналы данного изображения в формате `png` с заданными весами.

В невекторизованном решении результат вычисляется в цикле, как сумма матриц с заданными весами (суммирование матриц также производится в цикле). В векторизованном решении результат вычисляется с помощью функции `numpy.average()`. В функциональном решении результат получается с помощью функции `reduce` для списка из кортежей, составленных из весов и каналов изображения.

Для этой задачи было проведено три эксперимента с картинками разных размеров (m, n). Каждый эксперимент для первых двух картинок (меньших размеров) был повторен 100 раз для векторизованного и функционального решения, и 10 раз для

Параметры задачи		Время работы		
m	n	Невект. реализация	Вект. реализация	Функ. реализация
481	209	343 ms	9.95 ms	3.21 ms
960	804	1.41 s	39.9 ms	10.9 ms
1800	2880	8.57 s	274 ms	78.1 ms

Таблица 5: Результаты экспериментов по пятой задаче

невекторизованного. Для самой большой картинки векторизованное и функциональное решения запускались по 10 раз, а невекторизованное — 2 раза, после чего результаты усреднялись. Результаты экспериментов приведены в таблице 5.

2.6 Задача 6

В этой задаче требовалось реализовать кодирование длин серий для данного вектора $v \in \mathbb{R}^l$.

В невекторизованном решении списки элементов кодирующих векторов генерируются в цикле по всем элементам данного вектора. Векторизованное решение приведено в листинге 2. Функциональное решение приведено в листинге 3.

Листинг 2: Векторизованное решение задачи 6

```
def vec_run_length_encoding(vector):
    vec = vector + 1
    elems = vec[np.roll(vec, 1) != vec]
    if (elems.size == 0):
        return np.array([vec[0] - 1]), np.array([vec.size])
    reverse_cumsum = (np.cumsum(vec[::-1]))[::-1]
    elem_reverse_cumsum = reverse_cumsum[np.roll(vec, 1) != vec]
    denom = np.copy(elems)
    denom[elems == 0] = 1
    elem_count = (elem_reverse_cumsum -
                  np.hstack((np.roll(elem_reverse_cumsum, -1)[:-1],
                  0))) / denom
    return elems-1, elem_count.astype(int)
```

Для данной задачи было проведено три эксперимента для различных l. Результаты экспериментов приведены в таблице 6.

Параметры задачи	Время работы		
l	Невект. реализация	Вект. реализация	Функ. реализация
30	20.4 μs	70 μs	58.5 μs
1000	518 μs	105 μs	1.53 ms
10000	5.34 ms	366 μs	16.4 ms

Таблица 6: Результаты экспериментов по шестой задаче

Листинг 3: Функциональное решение задачи 6

```
def fun_run_length_encoding(vec):
    def fun(i, elem):
        if (i >= len(vec)) or (vec[i] != elem):
            return 0
        return 1 + fun(i + 1, elem)

    elems = np.array([vec[0]] + [elem for elem, prev in
                                 list(zip(vec[1:], vec[:-1]))
                                 if elem != prev])
    indices = [0] + [i for i in range(1, len(vec))
                     if vec[i] != vec[i-1]]
    return (elems,
            np.array([fun(i, elem) for i, elem in
                      list(zip(indices, elems))]))
```

2.7 Задача 7

В этой задаче требовалось вычислить матрицу попарных евклидовых расстояний между строкамми матриц $X \in \mathbb{R}^{n \times d}$ и $Y \in \mathbb{R}^{m \times d}$. Также предлагалось сравнить результаты и скорость работы реализаций со стандартной реализацией `scipy.spatial.distance.cdist`.

В невекторизованном решении данной задачи требуемая матрица вычисляется с помощью двух вложенных циклов по объектам первой и второй выборки (норма разности между векторами также вычисляется в цикле). Векторизованное решение основывется на использовании *broadcasting* — в каждую из выборок добавляется новая ось на свое место, после чего вычисляется норма их разности по третьей оси: `np.linalg.norm((x.T)[:, :, None] - (y.T)[:, None, :], axis=0)`. В функциональном решении требуемая матрица строится с помощью *list comprehension*.

Время работы для $(d, n, m) = (2, 5, 10)$ усреднялось по 1000 запусков для всех методов. Время работы для $(d, n, m) = (30, 100, 50)$ усреднялось по 100 запускам для всех методов. В последнем же эксперименте время работы для всех методов, кроме

Параметры задачи			Время работы			
d	n	m	Невект. реализация	Вект. реализация	Функ. реализация	Станд. реализация
2	5	10	230 μs	20.5 μs	577 μs	26.1 μs
30	100	50	149 ms	470 μs	48.6 ms	178 μs
100	500	1000	49 s	478 ms	4.94 s	43.5 ms

Таблица 7: Результаты экспериментов по седьмой задаче

невекторизованного усреднялось по 10 запускам, а для невекторизованного — по 2. Результаты экспериментов приведены в таблице 7.

2.8 Задача 8

В этой задаче требовалось вычислить значения плотности многомерного распределения $\mathcal{N}(m, cov)$, где $m \in \mathbb{R}^d$, $cov \in \mathbb{R}^{d \times d}$ в данных точках $X \in \mathbb{R}^{n \times d}$. Также предлагалось сравнить результаты и скорость работы реализаций со стандартной реализацией `scipy.stats.multivariate_normal(m, C).logpdf(X)`.

В невекторизованном решении плотность подсчитывается для каждой точки в цикле. При этом произведение матриц реализовано с помощью вложенных циклов. В векторизованном решении в формулу для плотности нормального распределения в данной точке x вместо одной точки подставляется матрица X. В качестве ответа возвращается диагональ полученной таким образом матрицы. В функциональном решении вектор плотностей строится так же, как и в невекторизованном варианте, только вместо циклов используется *list comprehension*, и для умножения матриц используется `numpy.dot()`.

В экспериментах для $(d, n) = (30, 50)$ результаты усреднялись по 1000 запускам для всех реализаций, кроме невекторизрванной, и по 100 запускам для невекторизованной. В экспериментах для $(d, n) = (300, 100)$ результаты усреднялись по 100 запускам для всех реализаций, кроме невекторизрванной, и по 2 запускам для невекторизованной. Наконец, для $(d, n) = (1000, 1000)$ результаты усреднялись по 10 запускам для всех реализаций, кроме невекторизрванной, а для невекторизованной реализации эксперименты не проводились (время ее работы слишком велико). Результаты экспериментов приведены в таблице 8.

Параметры задачи		Время работы			
d	n	Невект. реализация	Вект. реализация	Функ. реализация	Станд. реализация
30	50	30.7 ms	151 μs	952 μs	381 μs
300	100	5.65 s	5.08 ms	12.1 ms	16.2 ms
1000	1000	—	197 ms	451 ms	320 ms

Таблица 8: Результаты экспериментов по восьмой задаче

3 Выводы

В данной работе было произведено сравнение трех подходов к написанию программ на `Python` — векторизованного, невекторизованного, и в стиле функционального программирования.

Векторизованное решение проиграло невекторизованному только в одном эксперименте — в шестой задаче на данных наименьшего размера. Результат этого эксперимента можно объяснить тем, что предложенное мной векторизованное решение данной задачи имеет высокую сложность (см. листинг 2), и поэтому на данных маленьких объемов более простая невекторизованная реализация выигрывает.

На данных средних и больших объемов векторизованная реализация во всех экспериментах существенно выигрывает у невекторизованной.

Функциональная реализация оказалась эффективнее векторизованной в четырех экспериментах — в первом эксперименте шестой задачи и во всех экспериментах пятой задачи. Объяснения результатов первого эксперимента шестой задачи были даны выше. Что же касается пятой задачи, разница между функциональным и векторизованным решением для нее заключалась в способе подсчета взвешенного среднего между каналами. По всей видимости, метод `numpy.average()` менее эффективен, чем `reduce()`, если требуется посчитать среднее небольшого количества точек.

Таким образом, на практике всегда лучше по возможности использовать векторизованную реализацию.