Отчет по первой лабораторной работе «Изучение Python, Numpy» по предмету практикум ЭВМ

Алескин Александр

9 октября 2015

Содержание

1 Введение

Цели данной работы:

1. Освоение языка Python и системы научных вычислений NumPy.

2. Проведение экспериментов с целью измерить скорости работы функций.

В рамках данного задания предполагалось решение восьми задач. В каждом задании предложено рассмотреть как минимум 3 разных варианта решения задачи, при этом один из вариантов должен быть полностью векторизованный, а другой полностью без векторизации. Ниже представлено решение задач и их анализ, сделаны соответствующие выводы.

Ввиду особенностей постановки задачи, прототипы функций для решения каждой задачи носят однотипные имена: *var_vec, var_non_vec, var_extra*, где первый вариант полностью векторизованный, второй — без использования векторизация, а третий чаще всего частично векторизованный и с использованием **list comprehension**.

Отчет о каждом задании составлен в формате:

- Анализ задачи

- Приведение фрагментов кода

- Анализ экспериментальных данных

В каждом фрагменте кода предполагается, что соответствующие библиотеки `NumPy` подключены, в частности библиотека `NumPy` подключена как `np`. Задание выполнено на ОС Ubuntu.

2 Задание 1

Условие задания: «Подсчитать произведение ненулевых элементов на диагонали прямоугольной матрицы.»

Задача делится на 3 этапа: нахождение диагональных элементов, выбор из них ненулевых и перемножение. Входные данные: матрица narray X, размерности 2. Выход: число. Векторный вариант реализован был следующим образом:

```
diag = X.diagonal()
return np.multiply.reduce(diag[diag != 0])
```

Невекторизованный вариант:

```
x = X.shape[0]
y = X.shape[1]
if x > y:
    min_sz = y
else:
    min_sz = x
sum_val = 1
for i in range(min_sz):
    if X[i,i] != 0:
        sum_val = sum_val * int(X[i,i])
return sum_val
```

Третий вариант с использованием list comprehension:

```
min_sz = min((X.shape))
mass = [X[i,i] for i in range(min_sz) if X[i,i] != 0]
return np.multiply.reduce(mass)
```

Результаты эксперимента,приведенные на рис. 1, подтверждают, что векторизованная функция работает быстрее на порядок, так как задача делиться на выполнение функций, написанные на языке C. List Comprehension незначительно ускоряет работу функции.

3 Задание 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]], . . . , X[i[N-1], j[N-1]]])» . Векторный вариант реализован был следующим образом:

```
return X[i,j].copy()
```

Невекторизованный вариант:

```
Y = []
for k in range(len(i)):
    Y += [X[i[k], j[k]]]
return np.array(Y)
```

Третий вариант с использованием list comprehension:

```
np.array([X[i[k],j[k]] for k in range(len(i))])
```

Результаты эксперимента для второй (рис 2) задачи аналогичны результатам первой. Стоит отметить, что зависимость векторизованного метода от размера вектора индексации почти отсутствует.

4 Задание 3

Условие:«Даны два вектора x и y. Проверить, задают ли они одно и то же мультимножество». Данная задача имеет несколько простых решений с использованием библиотеки NumPy. Векторные варианты:

1. вариант с функцией np.unique:

```
val = False
if x.size == y.size:
    x_val, x_count = np.unique(x, return_counts = True)
    y_val, y_count = np.unique(y, return_counts = True)
    if len(x_val) == len(y_val):
        val = (x_val == y_val).all()
        if val:
            val &= (x_count == y_count).all()
return val
```

2. вариант сортировки массива по порядку:

```
val = False
if x.size == y.size:
    a = x.copy()
    b = y.copy()
    a.sort()
    b.sort()
    if a.size == b .size:
        if (a == b).all() :
            val = True
return val
```

3. вариант с использованием np.bincount:

```
val = False
if x.size == y.size:
    x_b = np.bincount(x)
    y_b = np.bincount(y)
    if (x_b.size == y_b.size):
        if (x_b == y_b).all():
            val = True
return val
```

Невекторизованный вариант повторяет работу 1 варианта, но без использование функции np.unique.

Из результатов эксперимента (рис 3) следует, что

- np.bincount решает задачу быстрее, хотя и требует больше памяти
- все векторизованные варианты работают значительно быстрее не векторизованного (то, что на малом размере np.unique работает медленнее не векторизованного обусловлено медленной работой функции в-общем и тем, что когда мультимножества не равны, функция var_non_vec не выполняет большую часть вычислений, что нельзя избежать в векторном случае).

5 Задание 4

Условие задания: «Найти максимальный элемент в векторе x среди элементов, перед которыми стоит нулевой.» Задача делится на 2 этапа: выбор нужных элементов, нахождение максимума среди выбранных элементов. Входные данные: вектор narray x. Выход: число. Векторный вариант реализован был следующим образом:

```
maska = np.roll(x == 0, 1)
maska[0] = False
return np.amax(x[maska])
```

Невекторизованный вариант:

```
val_max = 0
for i in range(1, x.size):
    if (x[i-1] == 0) & (x[i] > val_max):
        val_max = x[i]
return val_max
```

Третий вариант с частичной векторизацией:

```
val_max = 0
try:
    for i in np.where(x == 0)[0]:
        if x[i+1] > val_max:
            val_max = x[i+1]
except IndexError:
    pass
return val_max
```

Результаты эксперимента,приведенные на рис. 4, подтверждают, что векторизованная функция работает быстрее, основное время выполнения функции занимает прохождение по циклу.

6 Задание 5

Условие задания: «Дан трёхмерный массив, содержащий изображение, а также вектор длины numChannels. Сложить каналы изображения с указанными весами, и вернуть результат в виде матрицы размера (height, width).»

Так как предполагаются изображения формата .png, то формат ввода и вывода ячеек изображения uint8. Векторный вариант реализован был следующим образом:

```
return np.uint8(np.round(np.sum(Image*numChannels, axis = 2)))
```

Невекторизованный вариант:

```
black = np.zeros(Image.shape[0:2], dtype= Image.dtype)
for i in np.arange(Image.shape[0]):
    for j in np.arange(Image.shape[1]):
        val  = 0.0
        for k in range(3):
            val += Image[i, j, k] * numChannels[k]
        black[i, j] =np.uint8(round(val))
return black
```

Третий вариант с частичной векторизацией:

```
black = np.zeros(Image.shape[0:2])
for k in range(3):
    black += Image[:,:,k]*numChannels[k]
return np.uint8(np.round(black))
```

Результаты эксперимента аналогичны предшествующим. Хотя в частичной векторизованной функции цикл лишь по третьему аргументу (по матрицам цвета), на рис. 5 заметна разница между векторизованным и не векторизованным алгоритмом.

7 Задание 6

Условие задания: «Реализовать кодирование длин серий (Run-length encoding). Дан вектор x. Необходимо вернуть кортеж из двух векторов одинаковой длины.»

Входные данные: вектор narray x. Выходные данные: вектора narray vals и repeats. Первый содержит каких чисел серии в векторе x, а второй — длины серий. Алгоритм решения представляет собой нахождение элементов вектора x отличных от предыдущих, а затем подсчета расстояния между ними.

Векторный вариант находит необходимые элементы как ненулевые элементы разности вектора x и сдвинутым на 1 позицию к началу вектора x, а длину серий, как расстояние между позициями этих чисел. Некомпактность кода следствие особых ситуаций в первом и конечном элементах вектора:

```
diff = x[1:] - x[:-1]
pos = np.where(diff != 0)[0] +1
reps = pos[1:] - pos[:-1]
vals = np.empty((pos.shape[0]+1), dtype = x.dtype)
vals[0] = x[0]
vals[1:] = x[pos].copy()
repeats = np.empty((pos.shape[0]+1), dtype = pos.dtype)
repeats[0] = pos[0] - 0
repeats[1:-1] = reps
repeats[-1] = x.size - pos[-1]
return (vals, repeats)
```

В отличие от векторизованного алгоритма невекторизованный прост и предельно ясен:

```
val   = x[0];
count = 0;
val_list = [];
count_list = [];
for i in x:
    if i == val:
        count += 1
    else:
        val_list.append(val)
        count_list.append(count)
        count = 1
        val = i
val_list.append(val)
count_list.append(count)
return (np.array(val_list), np.array(count_list))
```

Третий вариант с использованием list comprehension:

```
pos = [0] + [i for i in range(1,x.size) if x[i] != x[i-1]]
counts = [pos[i] - pos[i-1] for i in range(1, len(pos))] + \
         [x.size - pos[-1]]
return (x[np.array(pos)].copy(), np.array(counts))
```

Как видно из рис. 6 на малых размерах невекторизованный вариант работает быстрее. Это объясняется тем, что в невекторизованном варианте алгоритм написан под данную задачу, а не как в векторизованном варианте хитрая комбинация функций Numpy. Стоит отметить, что вариант с list comprehension работает медленнее невекторизованного, что можно объяснить существованием двух циклов в теле функции.

8 Задание 7

Условие задания: «Даны две выборки объектов - X и Y. Вычислить матрицу евклидовых расстояний между объектами.»

Пусть матрицы размера $m \times n$ и $k \times n$. Векторизованный вариант состоит в расширении матрицы X до размера $mk \times n$, так чтобы вся матрица повторялась k раз, а матрица У до тех же размером, но чтобы последовательно повторялись m раз строки матрицы. Считая евклидово расстояние между строками матриц, и воспользовавшись функцией reshape, получим искомую матрицу:

```
XX = np.repeat(X, Y.shape[0], axis = 0)
YY = np.tile(Y, (X.shape[0], 1))
Z = np.sum((XX-YY)**2, axis =1)**(0.5)
return np.reshape(Z, (X.shape[0], Y.shape[0]))
```

Невекторизованный вариант реализует наивный алгоритм подсчета расстояний:

```
Z = np.empty([X.shape[0], Y.shape[0]])
```

```
for i in range(X.shape[0]):
    for j in range(Y.shape[0]):
        val = 0.0
        for k in range(X.shape[1]):
            val += (X[i, k] - Y[j, k])**2
        Z[i,j] = val**(0.5)
return Z
```

Третий вариант это предложенная функция scipy.spatial.distance.cdist.

Результаты эксперимента на рис. 7. Как всегда, невекторизованый вариант работает на 1–2 порядка медленнее. Функция ж scipy.spatial.distance.cdist хороший пример оптимизации алгоритма.

9 Задание 8

Условие задания: «Реализовать функцию вычисления логарифма плотности многомерного нормального распределения Входные параметры: точки X, размер (N, D), мат. ожидание m, вектор длины D, матрица ковари- аций C, размер (D, D).»

Логарифм плотности многомерного нормального распределения вычисляется по формуле:

$$f(x) = \ln\left(\frac{1}{\sqrt{(2\pi)^n \mid C \mid}} e^{-\frac{1}{2}(x-m)^T C^{-1}(x-m)}\right) \tag{1}$$

Векторный вариант решения:

```
Ex = np.exp(np.sum((X - m).\
      dot(np.linalg.inv(C))*(X - m), axis = 1) * (-0.5))
alfa = 1 / (((2*np.pi)**C.shape[0] * np.linalg.det(C)) ** 0.5)
return np.log(Ex * alfa)
```

Невекторизованный вариант реализует наивный алгоритм вычисления:

```
Z = np.empty([X.shape[0], Y.shape[0]])
for i in range(X.shape[0]):
    for j in range(Y.shape[0]):
        val = 0.0
        for k in range(X.shape[1]):
            val += (X[i, k] - Y[j, k])**2
        Z[i,j] = val**(0.5)
return Z
```

Третий вариант это предложенная функция scipy.stats.multivariate_normal.

Результаты эксперимента на рис. 8. Как всегда, невекторизованый вариант работает на 1–2 порядка медленнее. О плохой реализации функции scipy.stats.multivariate_normal говорит, как скорость выполнения функции, так и точность — часто данные имеют ошибку в четырнадцатом знаке после запятой для типа float (точность измерялась по отношению к невекторизованному варианту).

10 Тестирование функций

Все функции тестировались по одному алгоритму(с небольшими изменениями к каждому задания). Исходные данные генерировались при помощи функций randint и rand из библиотеки NumPy. Ниже приведен код для тестирования функций из пятого(наиболее простой) задания:

```
import exersize5 as ex5
import time

sizes = np.array([100, 250, 500, 1000])
size1 = sizes
mass = []
for i in range(sizes.size):
    arr = np.random.randint(2**20, size = (sizes[i], sizes[i]+100, 3))
    mass += [arr]
vec = np.zeros((size1.size))
non_vec = np.zeros((size1.size))
extra = np.zeros((size1.size))
timer = np.zeros((3))

for i in range(size1.size):
    for j in range(100//((i+1)**2)):
        start = time.time()
        ex5.var_vec(mass[i])
        end = time.time()
        timer[0] += (end - start)

        start = time.time()
        ex5.var_non_vec(mass[i])
        end = time.time()
        timer[1] += end - start

        start = time.time()
        ex5.var_extra(mass[i])
        end = time.time()
        timer[2] += end - start
    timer /= j + 1
    vec[i] = timer[0]
    non_vec[i] = timer[1]
    extra[i] = timer [2]
```

11 Заключение

Проделанная работа позволяет более уверенно пользоваться библиотекой Numpy и другими средствами Python.

На основании вышеизложенного, имеем что:

- Векторизованный вариант работает на 1–2 порядка быстрее невекторизованного, а иногда еще быстрее
- List comprehension незначительно ускоряет работу программы
- Невекторизованный вариант бывает полезен для проверки работы запутанного векторизованного варианта решения задачи
- Читабельность кода и время его написания векторизованного решения простых задач сопоставим с невекторизованным, а зачастую превосходит по этим показателям
- Не все функции библиотеки SciPy реализованы оптимальным способом.

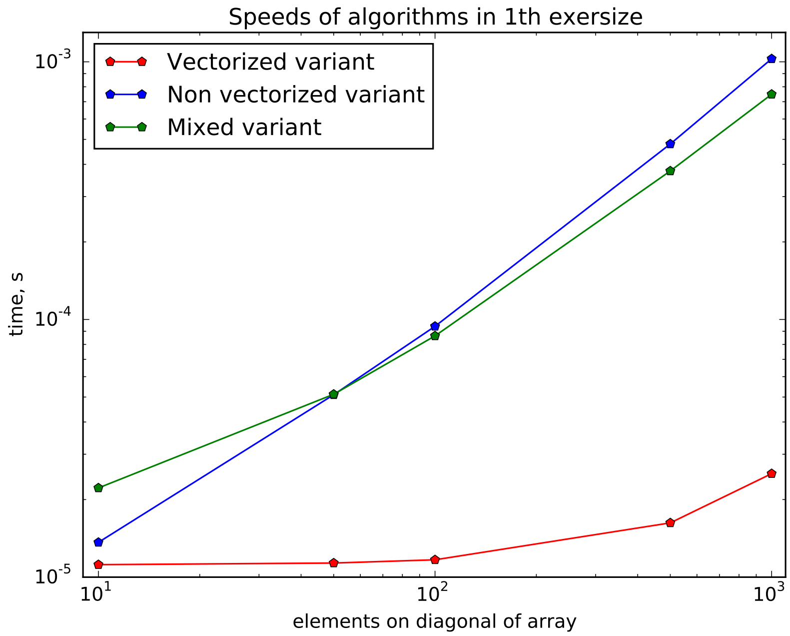

Рис. 1: Время работы алгоритмов задание №1

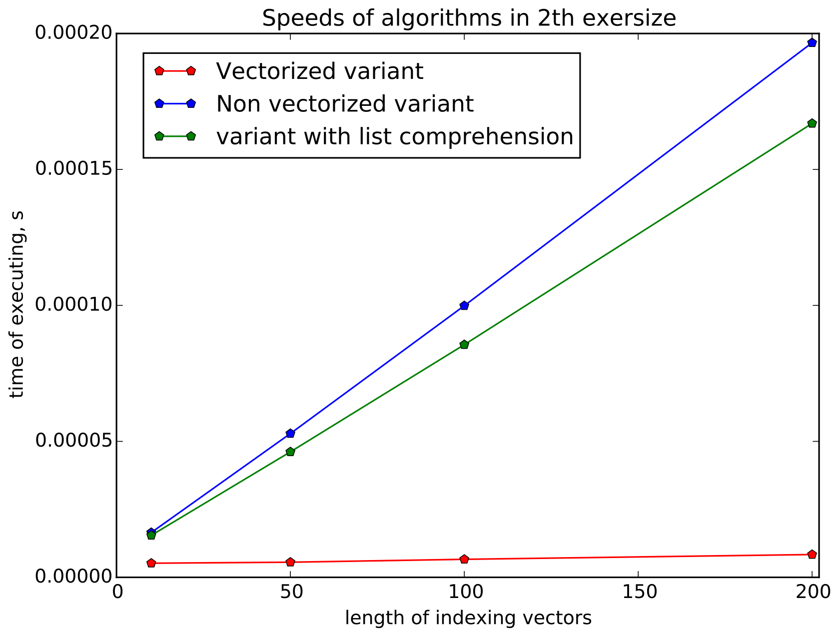

Рис. 2: Время работы алгоритмов задание №2

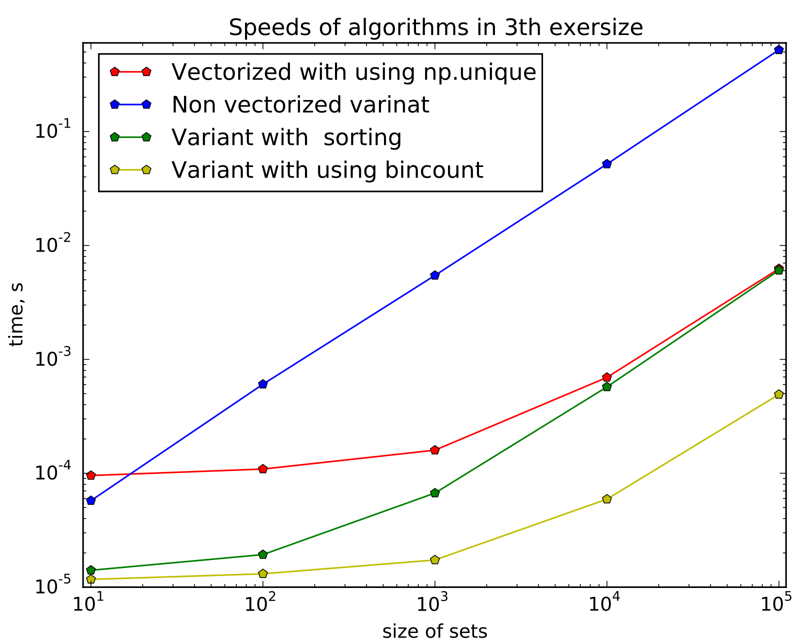

Рис. 3: Время работы алгоритмов задание №3

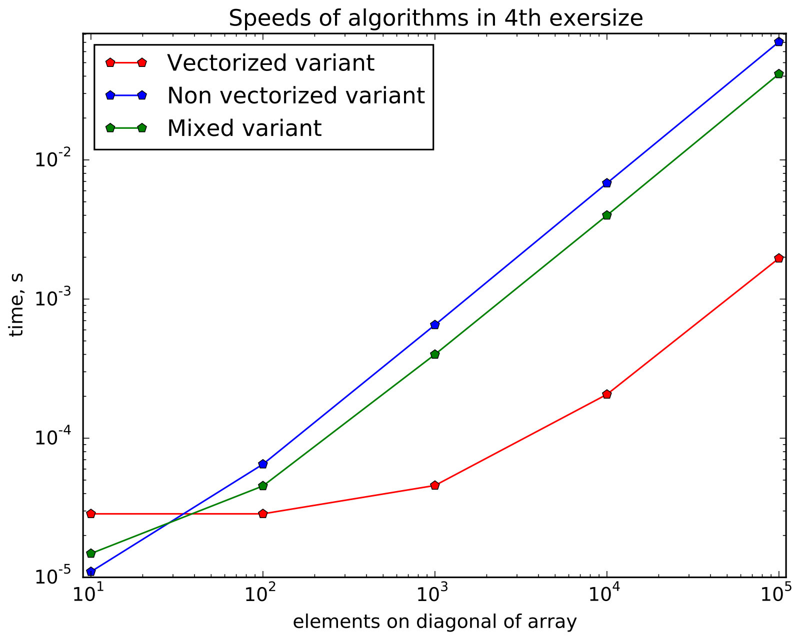

Рис. 4: Время работы алгоритмов задание №4

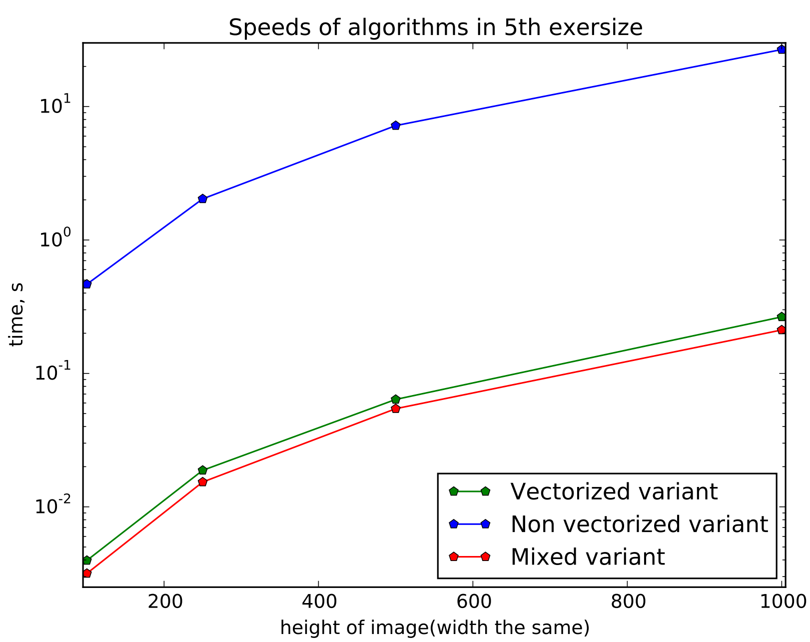

Рис. 5: Время работы алгоритмов задание №5

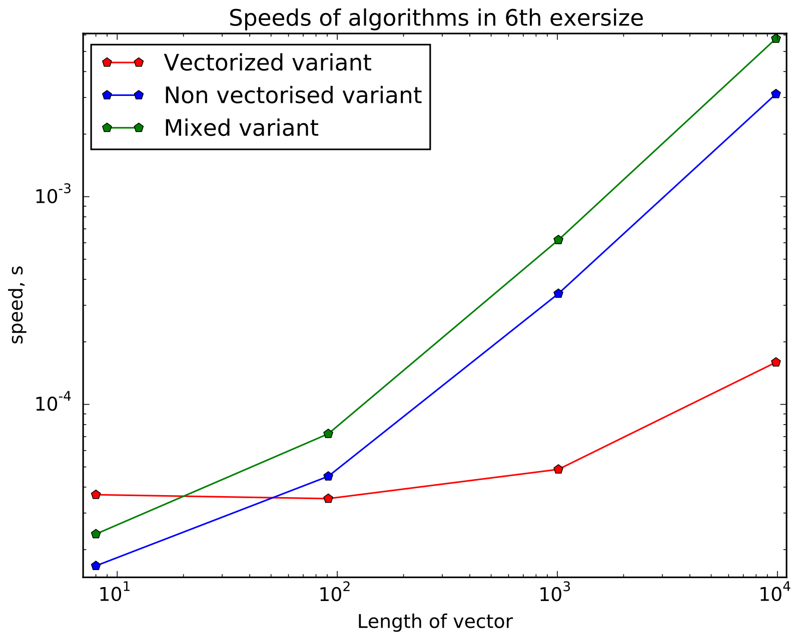

Рис. 6: Время работы алгоритмов задание №6

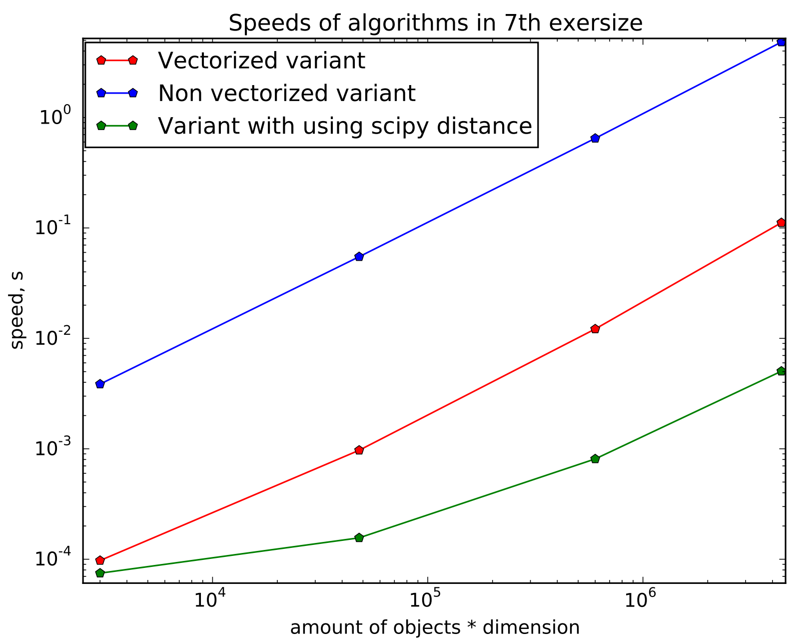

Рис. 7: Время работы алгоритмов задание №7

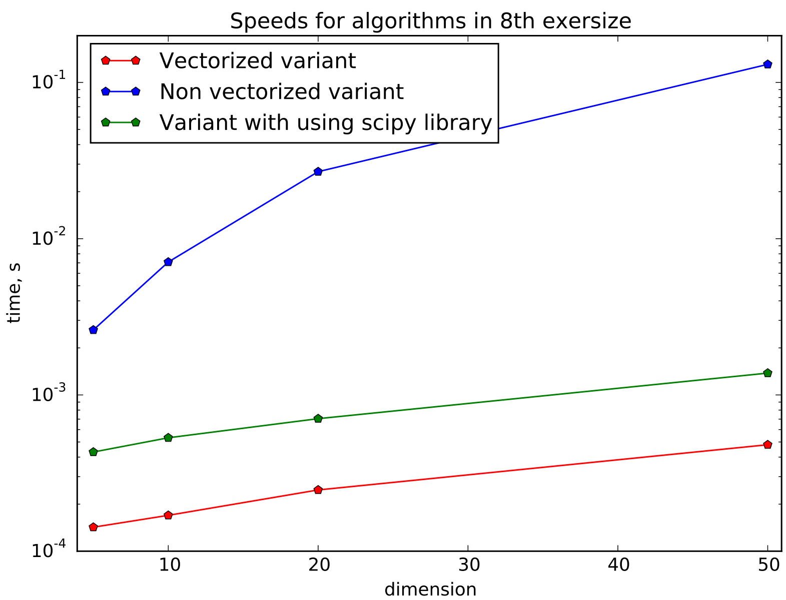

Рис. 8: Время работы алгоритмов задание №8