Московский государственный университет
имени М. В. Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Математических Методов Прогнозирования

# Отчет о выполнении задания №1 по практикуму на ЭВМ.

Выполнил студент 317 группы
*Гарипов Тимур Исмагилевич*

Москва, 9 октября 2015 г.

# Содержание

# 1 Описание задания

Данное задание направлено на освоение языка Python и системы научных вычислений NumPy.

Требуется для каждой из предложенных задач:

1. Написать на Python + NumPy несколько вариантов кода различной эффективности. Должно быть не менее трёх вариантов, в том числе как минимум один полностью векторизованный вариант и один вариант без векторизации. И третий альтернативный вариант решения, это может быть, например, наиболее хорошо читаемый способ решения или частично векторизованный вариант. Все варианты решения одной задачи должны содержаться в отдельном Python модуле.
2. Сравнить в IPython Notebook при помощи %timeit скорость работы на нескольких тестовых наборах разного размера (минимум 3).
3. Проанализировать полученные данные о скорости работы разных реализаций.
4. Получить выводы.

Для получения бонусных баллов за задание должны быть выполнены следующие требования:

- Написанный код полностью соответствует style guide PEP 8.
- Ко всем задачам присутствуют автоматические тесты, проверяющие совпадение результатов работы всех вариантов кода. Тесты должны использовать встроенный в Python фреймворк unittest.

## 2 Задачи

В описании всех задач предполагается, что модуль numpy импортирован под названием np.

### 2.1 Задача №1

**Постановка задачи**
Требуется подсчитать произведение ненулевых элементов на диагонали прямоугольной матрицы. Для X = np.array([[1, 0, 1], [2, 0, 2], [3, 0, 3], [4, 4, 4]]) ответ 3.

**Описание решений**

```python
def vectorized(X):
    """
    X --- 2d numpy array
    returns number
    """
    diag = np.diag(X)
    return np.prod(diag[np.nonzero(diag)])
```

**Листинг 1.1:** Векторизированный вариант

```python
def non_vectorized(X):
    """
    X --- 2d numpy array
    returns number
    """
    n = min(X.shape[0], X.shape[1])
    result = 1
    for i in range(n):
        if X[i, i] != 0:
            result *= X[i, i]
    return result
```

**Листинг 1.2:** Вариант без векторизации

```python
def alternative(X):
    """
    X --- 2d numpy array
    returns number
    """
    diag = np.diag(X).copy()
    diag[diag == 0] = 1
    return np.prod(diag)
```

**Листинг 1.3:** Альтернативный вариант

**Сравнение времени работы**
Для сравнения скорости работы методов генирировалась случайная X матрица размера msize. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 1.

| # | описание данных | Время работы (мс) | | |
|---|---|---|---|---|
| | | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | msize=(20, 30) | 0.019 | 0.029 | 0.028 |
| 2 | msize=(400, 400) | 0.032 | 0.525 | 0.038 |
| 3 | msize=(800, 600) | 0.038 | 0.824 | 0.041 |
| 4 | msize=(1500, 1500) | 0.072 | 2.058 | 0.064 |

**Таблица 1:** Задача №1. Сравнение времени работы

**Анализ результатов и выводы**
Как и ожидалось векторизованный и альтернативный варианты заметно превосходят вариант без векторизации. Причем различие в скорости работы вариантов, использующих numpy, не значительны.

Резльтаты свидетельствуют о том, что использование numpy даёт значительный выигрыш в скорости работы программы даже для простых задач.

## 2.2 Задача №2

**Постановка задачи**
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]]]).

**Описание решений**

```
def vectorized(X, i, j):
    """
    X --- 2d numpy array.
    i, j --- 1d numpy arrray of the same length
    returns --- 1d numpy array
    """
    return X[i, j]
```

**Листинг 2.1:** Векторизированный вариант

```
def non_vectorized(X, i, j):
    """
    X --- 2d numpy array.
    i, j --- 1d numpy arrray of the same length
    returns --- 1d numpy array
    """
    result = []
    for ind in range(len(i)):
        result += [X[i[ind], j[ind]]]
    return np.array(result)
```

**Листинг 2.2:** Вариант без векторизации

```python
def alternative(X, i, j):
    """
    X --- 2d numpy array
    i, j --- 1d numpy arrray of the same length
    returns --- 1d numpy array
    """
    return np.array([X[i[ind]][j[ind]] for ind in range(len(i))])
```

**Листинг 2.3:** Альтернативный вариант

### Сравнение времени работы

Для сравнения скорости работы методов генирировалась случайная X матрица размера msize и случайные векторы i и j длины n. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 2.

| | | Время работы (мс) | | |
|---|---|---|---|---|
| # | описание данных | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | msize=(50, 50), n=100 | 0.006 | 0.133 | 0.153 |
| 2 | msize=(500, 750), n=1000 | 0.022 | 1.302 | 1.483 |
| 3 | msize=(8000, 4000), n=8000 | 0.308 | 10.434 | 11.851 |

**Таблица 2:** Задача №2. Сравнение времени работы

### Анализ результатов и выводы

Результаты демонстрируют значительное превосходство векторизованного варианта решения. Эффективность двух друних вариантов практически эквивалентна.

Для данной задачи решение, использующее numpy не только значительно эффективнее, но и является наиболее лаконичным.

## 2.3 Задача №3

### Постановка задачи

Даны два вектора x и y. Проверить, задают ли они одно и то же мультимножество. Для x = np.array([1, 2, 2, 4]), y = np.array([4, 2, 1, 2]) ответ True.

### Описание решений

```python
def vectorized(x, y):
    """
    x, y --- 1d numpy array of integer numbers
    returns boolean
    """
    return np.all(np.sort(x) == np.sort(y))
```

**Листинг 3.1:** Векторизированный вариант

```python
def non_vectorized(x, y):
    """
    x, y --- 1d numpy array of integer numbers
    returns boolean
    """
    return sorted(list(x)) == sorted(list(y))
```

**Листинг 3.2:** Вариант без векторизации

```python
def alternative(x, y):
    """
    x, y --- 1d numpy arrays of integer numbers
    returns boolean
    """
    x_vals, x_cnt = np.unique(x, return_counts=True)
    y_vals, y_cnt = np.unique(y, return_counts=True)
    return np.all(x_vals == y_vals) and np.all(x_cnt == y_cnt)
```

**Листинг 3.3:** Альтернативный вариант

**Сравнение времени работы**

Для сравнения скорсти работы методов генерировался случайный массив размера n, содержащий целые числа. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 3.

| | | Время работы (мс) | | |
|---|---|---|---|---|
| # | описание данных | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | n=10000 | 1.635 | 15.917 | 1.930 |
| 2 | n=100000 | 18.876 | 189.952 | 20.358 |
| 3 | n=1000000 | 215.832 | 2398.368 | 230.888 |
| 4 | n=10000000 | 2588.981 | 29927.273 | 2895.783 |

**Таблица 3:** Задача №3. Сравнение времени работы

**Анализ результатов и выводы**

Невекторизованный вариант значительно отстает по производительности от двух других вариантов. Альтернативный вариант незначительно устапает в скорости векторизованному варианту. В данном случае идеи, положенные в основу решений, практически не отличаются, но тем не менее варианты, использующие numpy работают гораздо быстрее.

## 2.4 Задача №4

**Постановка задачи**

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

нулевой. Для x = np.array([6, 2, 0, 3, 0, 0, 5, 7, 0]) ответ 5.

### Описание решений

```python
def vectorized(x):
    """
    x --- 1d numpy array
    returns number
    """
    return np.max(x[np.minimum(np.where(x == 0)[0] + 1, len(x)-1)])
```

**Листинг 4.1:** Векторизированный вариант

```python
def non_vectorized(x):
    """
    x --- 1d numpy array
    returns number
    """
    result = -np.inf
    for i in range(1, len(x)):
        if x[i - 1] == 0:
            result = max(result, x[i])
    return result
```

**Листинг 4.2:** Вариант без векторизации

```python
def alternative(x):
    """
    x --- 1d numpy array
    returns number
    """
    result = -np.inf
    x = x[np.minimum(np.where(x == 0)[0] + 1, len(x)-1)]
    for value in x:
        result = max(result, value)
    return result
```

**Листинг 4.3:** Альтернативный вариант

Альтернативный вариант представляет собой частично векторизованное решение.

### Сравнение времени работы

Для сравнения скорсти работы методов генерировался случайный массив размера n, содержащий целые числа. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 4.

### Анализ результатов и выводы

Резльтаты экспериментов показывают, что частично векторизованный вариант в плане эффективности занимает промежутчное положение между векторизованным и невекторизованным вариантами.

| # | описание данных | Время работы (мс) | | |
|---|---|---|---|---|
| | | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | n=100000 | 1.456 | 75.897 | 20.796 |
| 2 | n=1000000 | 15.998 | 749.948 | 208.869 |
| 3 | n=10000000 | 184.911 | 7503.472 | 2128.060 |

**Таблица 4:** Задача №4. Сравнение времени работы

### 2.5 Задача №5

**Постановка задачи**

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). Преобразовать цветное изображение в оттенки серого, использовав коэффициенты np.array([0.299, 0.587, 0.114]).

**Описание решений**

```python
def vectorized(img, channelsWeights):
    """
    img --- 3d numpy array with shape=(ihHeight, inWidth, numChannels)
    channelsWeights --- 1d numpy array of size=numChannels
    returns 2d numpy array with shape=(inHeight, inWidth)
    """
    return np.sum(img * channelsWeights[np.newaxis, np.newaxis, :], axis=2)
```

**Листинг 5.1:** Векторизированный вариант

```python
def non_vectorized(img, channelsWeights):
    """
    img --- 3d numpy array with shape=(ihHeight, inWidth, numChannels)
    channelsWeights --- 1d numpy array of size=numChannels
    returns 2d numpy array with shape=(inHeight, inWidth)
    """
    result = np.zeros(img.shape[:2])
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            for k in range(img.shape[2]):
                result[i, j] += img[i, j, k] * channelsWeights[k]
    return result
```

**Листинг 5.2:** Вариант без векторизации

```python
def alternative(img, channelsWeights):
    """
    img --- 3d numpy array with shape=(ihHeight, inWidth, numChannels)
    channelsWeights --- 1d numpy array of size=numChannels
    returns 2d numpy array with shape=(inHeight, inWidth)
    """
    result = np.zeros(img.shape[:2])
    for k in range(img.shape[2]):
        result += img[:, :, k] * channelsWeights[k]
    return result
```

**Листинг 5.3:** Альтернативный вариант

Векторизованный вариант состоит из следующих шагов: умножение каждого канала на соответствующий вес (при этом используется broadcasting), суммирование полученных значений по всем каналам.

### Сравнение времени работы

Для сравнения скорсти работы методов генерировался случайный трехмерный массив размера imsize. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 5.

| # | описание данных | Время работы (мс) | | |
|---|---|---|---|---|
| | | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | imsize=(100, 100, 5) | 0.540 | 76.856 | 0.313 |
| 2 | imsize=(100, 100, 10) | 0.737 | 140.606 | 0.738 |
| 3 | imsize=(100, 100, 50) | 2.621 | 653.687 | 3.890 |
| 4 | imsize=(100, 100, 100) | 4.898 | 1302.950 | 8.048 |
| 5 | imsize=(200, 200, 4) | 2.014 | 254.366 | 0.853 |
| 6 | imsize=(200, 200, 8) | 2.918 | 461.242 | 2.958 |
| 7 | imsize=(200, 200, 12) | 3.837 | 663.771 | 6.159 |

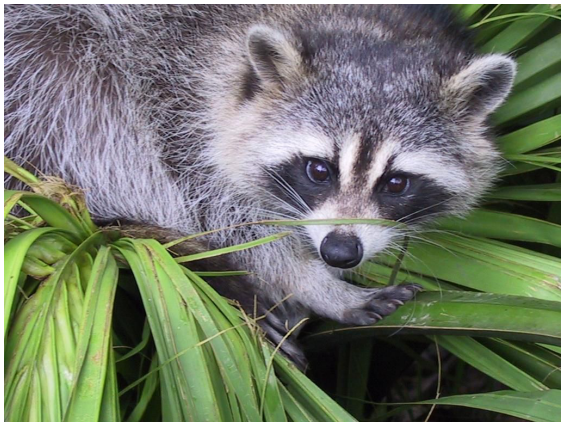
**Таблица 5:** Задача №5. Сравнение времени работы

На рисунке 1 представлен результат преобразования цветного изображения в оттенки серого.

### Анализ результатов и выводы

Во всех экспериментах вариант без векторизации выполняется за время, на 2 порядка превосходящее, время выполнения двух других вариантов.

Альтернативный вариант имеет преимущетсво перед векторизованным в тех случаях, в которых количество каналов в изображении мало. Но при увеличении количетсва каналов векторизованный вариант становится предподчительнее.



**(a)** Исходное изображение

**(b)** Результат работы

**Рис. 1:** Задача №5. Преобразования цветного изображения в оттенки серого

### 2.6 Задача №6

**Постановка задачи**

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. Пример: x = np.array([2, 2, 2, 3, 3, 3, 5]). Ответ: (np.array([2, 3, 5]), np.array([3, 3, 1])).

**Описание решений**

```
1  def vectorized(x):
2      """
3      x --- 1d numpy array of integer numbers
4      returns tuple of two 1d numpy arrays of integers number
5      """
6      pos = np.append(np.where(np.diff(x) != 0)[0], len(x)-1)
7      return (x[pos], np.diff(np.insert(pos, 0, -1)))
```

**Листинг 6.1:** Векторизированный вариант

```
def non_vectorized(x):
    """
    x --- 1d numpy array of integer numbers
    returns tuple of two 1d numpy arrays of integers number
    """
    values = [x[0]]
    cnt = []
    cur = 1
    for i in range(1, len(x)):
        if x[i] != x[i - 1]:
            cnt.append(cur)
            cur = 1
            values.append(x[i])
        else:
            cur += 1
    cnt.append(cur)
    return (np.array(values), np.array(cnt))
```

**Листинг 6.2:** Вариант без векторизации

```
def alternative(x):
    """
    x --- 1d numpy array of integer numbers
    returns tuple of two 1d numpy arrays of integers number
    """
    pos = np.where(np.diff(x) != 0)[0]
    if (pos.size == 0):
        return (np.array([x[0]]), np.array(x.size))
    cnt = [pos[0] + 1]
    for i in range(1, len(pos)):
        cnt.append(pos[i]-pos[i-1])
    cnt.append(len(x) - pos[-1] - 1)
    values = x[pos]
    values = np.append(values, x[-1])
    return (values, np.array(cnt))
```

**Листинг 6.3:** Альтернативный вариант

Векторизованное решение устроенно следующим образом. Функция np.diff используется для нахождения позиций тех элементов массива, значение которых отличается от значения следующего элемента (считается, что последний элемент массива не совпадает со следующим за ним). Затем из массива выбираются элементы, находящиеся на найденных позициях, и с помощью np.diff вычисляется необходимое количетсво повторений каждого значения как разность между индексами элементов, имеющих несовпадающие значения.

Альтернативный вариант представляет собой частично векторизованное решение.

### Сравнение времени работы

Для сравнения скорсти работы методов генерировался случайный массив размера n, содержащий целые числа. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 6.

| # | описание данных | Время работы (мс) | | |
|---|---|---|---|---|
| | | векторизованный вариант | вариант без векторизации | альтернативный вариант |
| 1 | n=10000 | 0.201 | 11.978 | 6.131 |
| 2 | n=100000 | 1.588 | 146.886 | 86.245 |
| 3 | n=1000000 | 17.307 | 1463.759 | 898.774 |

**Таблица 6:** Задача №6. Сравнение времени работы

### Анализ результатов и выводы

Векторизованное решение оказывается заметно эффективнее других вариантов.

Так как в альтернативном варианте второй вызов функции np.diff из векторизованного варианта заменён на цикл, можно сделать вывод о том, что использование функции np.diff делает решение существено эффективнее.

## 2.7 Задача №7

### Постановка задачи

Даны две выборки объектов — X и Y. Вычислить матрицу евклидовых расстояний между объектами. Сравнить с функцией scipy.spatial.distance.cdist.

### Описание решений

```
def vectorized(X, Y):
    """
    X, Y --- 2d numpy array with the same shape[1]
    returns --- 2d numpy array with shape (X.shape[0], Y.shape[0])
    """
    return np.sum((X[:, np.newaxis, :] - Y[np.newaxis, :, :]) ** 2,
                  axis=2) ** 0.5
```

**Листинг 7.1:** Векторизованный вариант

```
def scipy_standart(X, Y):
    """
    X, Y --- 2d numpy array with the same shape[1]
    returns --- 2d numpy array with shape (X.shape[0], Y.shape[0])
    """
    return scipy.spatial.distance.cdist(X, Y)
```

**Листинг 7.2:** Стандартный вариант scipy

```python
def alternative(X, Y):
    """
    X, Y --- 2d numpy array with the same shape[1]
    returns --- 2d numpy array with shape (X.shape[0], Y.shape[0])
    """
    result = np.zeros((X.shape[0], Y.shape[0]))
    for i in range(X.shape[0]):
        result[i] = np.sum((X[i]-Y) ** 2, axis=1)
    return np.sqrt(result)
```

**Листинг 7.3:** Альтернативный вариант

В векторизованном варианте вычисления начинаются с нахождения с помощью broadcasting трехмерного тензора $Z[i, j, k] = (X[i, k] - Y[j, k])^2$. Затем происходит суммирование полученных значений по $k$ и вычисление квадратного корня.

В альтернативном варианте для каждой строки матрицы X с помощью broadcasting вычисляется её разность со всеми строками матрицы Y, затем разности возводятся во вторую степень и суммируются. Для получения ответа остаётся только извлечь квадратный корень из полученных значнений.

### Сравнение времени работы

Для сравнения скорсти работы методов генерировилсь случайные матрицы X и Y с размерами (n, d) и (m, d) соответственно. Время работы всех вариантов решения для входных данных различного объема представлено в таблице 7.

| | | Время работы (мс) | | |
|---|---|---|---|---|
| # | описание данных | векторизованный вариант | стандартный вариант scipy | альтернативный вариант |
| 1 | n=20, m=30, d=10 | 0.097 | 0.061 | 0.564 |
| 2 | n=400, m=400, d=2 | 11.009 | 2.593 | 18.966 |
| 3 | n=100, m=100, d=10 | 1.103 | 0.300 | 3.510 |
| 4 | n=100, m=100, d=100 | 8.048 | 2.216 | 8.166 |
| 5 | n=100, m=100, d=500 | 79.720 | 10.846 | 26.725 |
| 6 | n=100, m=100, d=1000 | 174.507 | 21.609 | 53.439 |
| 7 | n=400, m=400, d=20 | 31.504 | 7.642 | 33.158 |
| 8 | n=400, m=400, d=100 | 302.907 | 34.916 | 93.727 |
| 9 | n=400, m=400, d=400 | 720.613 | 138.863 | 408.538 |
| 10 | n=400, m=400, d=800 | 1372.627 | 284.279 | 974.328 |

**Таблица 7:** Задача №7. Сравнение времени работы

### Анализ результатов и выводы

Стандартный вариант scipy демонстрирует наилучшую эффективность. Векторизованный вариант оказывается лучше альтернативного при малых значениях d, но при

больших значениях устапает в скорости. Скорее всего это связано с тем, что альтернативный вариант при больших d использует меньшие объемы памяти.

## 2.8 Задача №8

### Постановка задачи

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.

### Описание решений

```python
def vectorized(X, m, C):
    """
    X --- NxD 2d numpy array
    m --- 1d numpy array of size D
    C --- DxD 2d numpy array
    returns 1d numpy array of size N
    """
    X = X - m[np.newaxis, :]
    d = len(m)
    invC = np.linalg.inv(C)
    logdet = np.linalg.slogdet(C)[1]
    const = -0.5 * (d * np.log(2.0 * np.pi) + logdet)
    pw = -0.5 * np.diag(np.dot(np.dot(X, invC), X.T))
    return const + pw
```

**Листинг 8.1:** Векторизованный вариант

```python
def scipy_standart(X, m, C):
    """
    X --- NxD 2d numpy array
    m --- 1d numpy array of size D
    C --- DxD 2d numpy array
    returns 1d numpy array of size N
    """
    return scipy.stats.multivariate_normal(m, C).logpdf(X)
```

**Листинг 8.2:** Стандартный вариант scipy

```python
def alternative(X, m, C):
    """
    X --- NxD 2d numpy array
    m --- 1d numpy array of size D
    C --- DxD 2d numpy array
    returns 1d numpy array of size N
    """
    X = X - m[np.newaxis, :]
    d = len(m)
    invC = np.linalg.inv(C)
    logdet = np.linalg.slogdet(C)[1]
    const = -0.5 * (d * np.log(2.0 * np.pi) + logdet)
    result = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        result[i] = const - 0.5 * np.dot(np.dot(X[i, :], invC), X[i, :].T)
    return result
```

**Листинг 8.3:** Альтернативный вариант

Векторизованное и альтернативное решения вычисляют требуемые значения, используя формулу:

$$\log p(x) = -\frac{d\log(2\pi) + \log|C| + (x-m)C^{-1}(x-m)^T}{2},$$

где $x$ — вектор-строка длины $d$, содержащий координаты точки, в которой требуется вычислить логарифм плотности; $m$ — вектор-строка длины $d$, содержащий математические ожидания и $C$ матрица ковариаций $d \times d$.

Но в векторизоввнном решении последнее слагаемое в числителе дроби вычисляется, при подставлении вместо x выражения $(x-m)$ и $(x-m)^T$ всех возможных пар строк матрицы $X$, после чего из полученных значений выбираются те, которые вычислены для пар, в которых участвовала одна и та же строка.

### Сравнение времени работы

Для сравнения скорсти работы методов генерировились случайная матрица X размера (n, d); случайные вектор m размера d и случайная симметричная положительно определенная матрица C размера (d, d). Время работы всех вариантов решения для входных данных различного объема представлено в таблице 8.

Тестовые запуски показали, что резльтаты работы методов совпадают с абсолютной точностью $10^{-5}$.

### Анализ результатов и выводы

Стандартный вариант scipy демонтсрирует наилучшее время работы. Векторизованный вариант работает быстрее альтернативного лишь при достаточно малых значениях n и d, так как в этом варианте происходят «лишние» вычисления.

| # | описание данных | Время работы (мс) | | |
|---|---|---|---|---|
| | | векторизованный вариант | стандартный вариант scipy | альтернативный вариант |
| 1 | n=100, d=10 | 0.484 | 0.589 | 1.105 |
| 2 | n=100, d=80 | 6.080 | 5.845 | 4.986 |
| 3 | n=100, d=100 | 9.354 | 9.189 | 7.876 |
| 4 | n=100, d=400 | 306.027 | 264.812 | 297.941 |
| 5 | n=200, d=10 | 1.443 | 0.594 | 1.937 |
| 6 | n=200, d=100 | 18.971 | 11.569 | 11.097 |
| 7 | n=200, d=200 | 71.188 | 53.179 | 54.177 |
| 8 | n=200, d=800 | 2684.795 | 1951.876 | 2553.240 |
| 9 | n=2000, d=2 | 56.682 | 0.573 | 17.207 |
| 10 | n=2000, d=20 | 197.693 | 2.964 | 19.602 |
| 11 | n=2000, d=100 | 980.057 | 54.215 | 71.467 |
| 12 | n=2000, d=200 | 2148.684 | 216.427 | 248.340 |

**Таблица 8:** Задача №8. Сравнение времени работы

## 3 Анализ проделанной работы

Все задания, описанные в секции 1 выполнены.

- Для каждой задачи в отдельном модуле написаны три решения, кроме того в модуле для каждой задачи реализована функция, генерерующая входные данные для тестовых запусков.
- Также для каждой задачи подготовлен модуль, содержащий автоматические тесты для проверки совпадения результатов решений.
- В IPython notebook реализована функция для автоматического запуска экспериментов по задаче, которая использовалась для сравнения времени работы решений на различных входных данных.
- Коды всех решений, тестов и вспомогательных функций соответствуют style guide PEP 8.
- Проведен анализ полученных данных и получены выводы.
- О проделанной работе подготовлен данный отчёт.

## 4 Заключительные выводы

Данная работа, показывает что использование библиотеки numpy позволяет значительно повысить эффективность работы программы. Однако, имееют место случаи, в которых заранее не очевидно какой именно способ решения позволит добиться максимальной производительности. Полученные результаты свидетельствуют о том, что в таких случаях стоит выбирать способ решения исходя из ограничений на входные данные в конкретной задаче.