Министерство образования Республики Беларусь

УО «Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №10

По дисциплине: “Языки программирования”

Тема: “Изучение NumPy. Сравнение производительности с классическими библиотеками Python”

Выполнила:

Савицкая А.Ю., ПО-7

Проверил:

Бойко Д.О.

Брест,2021

**Ход работы:**

1. Для написания кода использовать библиотеки классического Python, NumPy и SciPy.
2. Код демонстрируется в Jupyter Notebook
3. По каждому заданию должно быть предоставлено не менее 3-х вариантов решения, среди которых:

* чистый NumPy (максимально оптимизованный, векторизованный)
* любой не векторизованный вариант
* любой другой вариант, желательно конкурентноспособный

1. Все варианты решения должны быть протестированы на скорость выполнения при помощи %timeit
2. Полученные результаты отразить в отчете и сделать выводы о производительности и комфорте использования NumPy в различных задачах.name:

**Задание 1:**

Подсчитать произведение ненулевых элементов на диагонали прямоугольной матрицы.

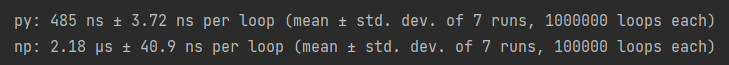
Пример: x = np.array([[1, 0, 1], [2, 0, 2], [3, 0, 3], [4, 4, 4]]) 

**Задание 2:**

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]]]).  
Пример:

x = [[9 4 2], [6 0 0], [9 9 3]]

i: [1 2 1]

j: [1 0 1]

**Задание 3:**

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

Пример: x = np.array([1,2, 2, 4]), y = np.array([4, 2, 1, 2])



**Задание 4:**

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

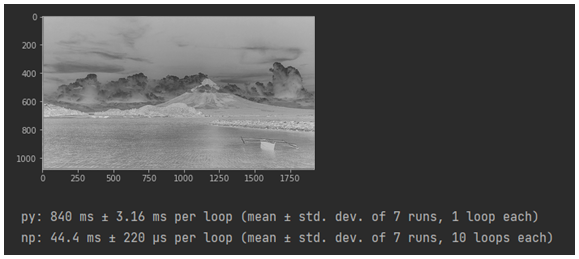
Пример: x = np.array([6, 2, 0, 3, 0, 0, 5, 7, 0]) 

**Задание 5:**

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). Считать реальное изображение можно при помощи функции scipy.misc.imread (если изображение не в формате png, установите пакет pillow: conda install pillow).  
Преобразуйте цветное изображение в оттенки серого, использовав коэффициенты np.array([0.299, 0.587, 0.114]).

Пример:



****

**Задание 6**

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])).

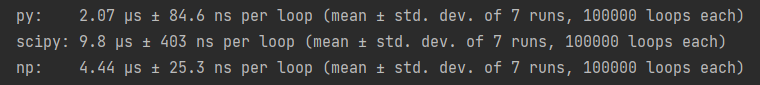
**Задание 7**

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

Пример:

x: [2 7 6 6 9 6 3 4 9]

y: [1 0 0 7 2 2 4 3 0]

Ответ: 15.329709716755891

**Задание 8**

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). Разрешается использовать библиотечные функции для подсчета определителя матрицы, а также обратной матрицы, в том числе в невекторизованном варианте. Сравнить с scipy.stats.multivariate\_normal(m, C).logpdf(X) как по скорости работы, так и по точности вычислений

**Код программы:**

**from** collections **import** Counter

**from** math **import** sqrt

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** scipy

**from** scipy.sparse.linalg **import** spsolve

**from** scipy.stats **import** multivariate\_normal

In [30]:

*# Task 1*

arr1\_py **=** [[1, 0, 1], [2, 0, 2], [3, 0, 3], [4, 4, 4]]

arr1\_np **=** np**.**array(arr1\_py)

**def** diagonal\_product\_py(arr):

product **=** 1

**for** i **in** range(len(arr[0])):

el **=** arr[i][i]

**if** el:

product **\*=** el

**return** product

**def** diagonal\_product\_np(arr):

diagonal **=** np**.**diag(arr)

**return** np**.**prod(diagonal[diagonal **!=** 0])

print("py:", end**=**" ")

**%timeit** diagonal\_product\_py(arr1\_py)

print("np:", end**=**" ")

**%timeit** diagonal\_product\_np(arr1\_np)

py: 422 ns ± 52.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

np: 7.01 µs ± 85.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [31]:

*# Task 2*

arr2\_py **=** [[9, 4, 2], [6, 0, 0], [9, 9, 3]]

i2\_py **=** [1, 2, 1]

j2\_py **=** [1, 0, 1]

arr2\_np **=** np**.**array(arr2\_py)

i2\_np **=** np**.**array(i2\_py)

j2\_np **=** np**.**array(j2\_py)

**def** vector\_py(arr, i, j):

vector **=** []

**for** k **in** range(len(i)):

vector**.**append(arr[i[k]][j[k]])

**return** vector

**def** vector\_np(arr, i, j):

**return** arr[i,j]

print("py:", end**=**" ")

**%timeit** vector\_py(arr2\_py, i2\_py, j2\_py)

print("np:", end**=**" ")

**%timeit** vector\_np(arr2\_np, i2\_np, j2\_np)

py: 485 ns ± 3.72 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

np: 2.18 µs ± 40.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [32]:

*# Task 3*

x3\_py **=** [1, 2, 2, 4]

y3\_py **=** [4, 2, 1, 2]

x3\_np **=** np**.**array(x3\_py)

y3\_np **=** np**.**array(y3\_py)

**def** equal\_py(arr1, arr2):

**return** sorted(arr1) **==** sorted(arr2)

**def** equal\_np(arr1, arr2):

**return** np**.**array\_equal(np**.**bincount(arr1), np**.**bincount(arr2))

print("py:", end**=**" ")

**%timeit** equal\_py(x3\_py, y3\_py)

print("np:", end**=**" ")

**%timeit** equal\_np(x3\_np, y3\_np)

py: 344 ns ± 3.81 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

np: 5.57 µs ± 476 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [33]:

*# Task 4*

x4\_py **=** [6, 2, 0, 3, 0, 0, 5, 7, 0]

x4\_np **=** np**.**array(x4\_py)

**def** null\_max\_py(arr):

count **=** 0

**for** i **in** range(1, len(arr)):

**if** **not** arr[i**-**1] **and** arr[i] **>** count:

count **=** arr[i]

**return** count

**def** null\_max\_np(arr):

*# zero\_indexes = np.where(arr == 0)*

zero\_indexes **=** arr **==** 0

**return** arr[1:][zero\_indexes[:**-**1]]**.**max()

print("py:", end**=**" ")

**%timeit** null\_max\_py(x4\_py)

print("np:", end**=**" ")

**%timeit** null\_max\_np(x4\_np)

py: 637 ns ± 53.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

np: 3.03 µs ± 24.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [34]:

*# Task 5*

photo\_arr **=** plt**.**imread("img.png")

photo\_arr\_py **=** photo\_arr**.**tolist()

photo\_arr\_np **=** np**.**array(photo\_arr\_py)

rgb\_ration\_py **=** [0.2989, 0.587, 0.114]

rgb\_ration\_np **=** np**.**array(rgb\_ration\_py)

**def** gray\_py(arr, rgb\_ration):

result\_arr **=** []

**for** height **in** arr:

new\_width **=** []

**for** width **in** height:

color **=** 0

**for** i **in** range(3):

color **+=** width[i] **\*** rgb\_ration[i]

new\_width**.**append(color)

result\_arr**.**append(new\_width)

**return** result\_arr

**def** gray\_np(arr, rgb\_ration):

**return** np**.**dot(arr[**...**,:3], rgb\_ration)

arr\_py **=** gray\_np(photo\_arr, rgb\_ration\_py)

plt**.**imshow(arr\_py, cmap**=**'gray')

plt**.**show()

print("py:", end**=**" ")

**%timeit** gray\_py(photo\_arr\_py, rgb\_ration\_py)

print("np:", end**=**" ")

**%timeit** gray\_np(photo\_arr\_np, rgb\_ration\_np)

py: 840 ms ± 3.16 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

np: 44.4 ms ± 220 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [35]:

*# Task 6*

arr6\_py **=** [2, 2, 2, 3, 3, 3, 5]

arr6\_np **=** np**.**array(arr6\_py)

**def** counts\_py(arr):

**return** tuple(zip(**\***Counter(arr)**.**items()))

**def** counts\_np(arr):

bin\_count **=** np**.**bincount(arr)

non\_zero\_indexes **=** np**.**nonzero(bin\_count)[0]

**return** non\_zero\_indexes, bin\_count[non\_zero\_indexes]

print("py:", end**=**" ")

**%timeit** counts\_py(arr6\_py)

print("np:", end**=**" ")

**%timeit** counts\_np(arr6\_np)

py: 1.85 µs ± 27 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

np: 2.21 µs ± 30.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [36]:

*# Task 7*

x7\_py **=** [2, 7, 6, 6, 9, 6, 3, 4, 9]

y7\_py **=** [1, 0, 0, 7, 2, 2, 4, 3, 0]

x7\_np **=** np**.**array(x7\_py)

y7\_np **=** np**.**array(y7\_py)

**def** euclid\_py(arr\_x, arr\_y):

**return** sqrt(sum((x **-** y) **\*\*** 2.0 **for** x, y **in** zip(arr\_x, arr\_y)))

**def** euclid\_np(arr\_x, arr\_y):

**return** np**.**linalg**.**norm(arr\_x **-** arr\_y)

print("py: ", end**=**" ")

**%timeit** euclid\_py(x7\_py, y7\_py)

print("scipy:", end**=**" ")

**%timeit** scipy.spatial.distance.euclidean(x7\_py, y7\_py)

print("np: ", end**=**" ")

**%timeit** euclid\_np(x7\_np, y7\_np)

py: 2.07 µs ± 84.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

scipy: 9.8 µs ± 403 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

np: 4.44 µs ± 25.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [37]:

*# Task 8*

sigma **=** np**.**array([

[2.3, 0, 0, 0],

[0, 1.5, 0, 0],

[0, 0, 1.7, 0],

[0, 0, 0, 2]

])

mu **=** np**.**array([2, 3, 8, 10])

x **=** np**.**array([2.1, 3.5, 8, 9.5])

**def** logarithm(x, mu, S):

norm\_coefficient **=** len(S) **\*** np**.**log(2 **\*** np**.**pi) **+** np**.**linalg**.**slogdet(S)[1]

err **=** x **-** mu

numerator **=** spsolve(S, err)**.**T**.**dot(err)

**return** **-**0.5 **\*** (norm\_coefficient **+** numerator)

print("np: ", end**=**" ")

**%timeit** logarithm(x, mu, sigma)

print("scipy:", end**=**" ")

**%timeit** multivariate\_normal(mu, sigma).logpdf(x)

np: 195 µs ± 834 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

scipy: 149 µs ± 2.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)