EPFL

HPC for numerical methods and data analysis

Fall Semester 2023

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 1 – September 10, 2024

Matrices and vectors in Python and MPI

Exercise I Matrices in Python

Find two efficient ways in Python to assign the following matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix},$$

without entering manually each element (*hint*: create two row vectors and then combine them to form a matrix).
Suitably use the Python commands to:

a) extract the element in the first row, third column of A;

b) extract the entire second row of A;

c) extract the first two columns of A;

d) extract the vector containing all the elements of the second row of A except for the third element.

Modify your code accordingly to assign the following matrices as well, calculate the time it takes for Python to assign them in both your methods. Which method is better? Compare your results with one of your classmates.

$$M_1 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \\ 19 & 20 & 21 \\ 22 & 23 & 24 \\ 25 & 26 & 27 \\ 28 & 29 & 30 \\ 31 & 32 & 33 \end{bmatrix}$$

$$M_4 = \begin{bmatrix} 1 & 2 & \dots & 250 \\ \vdots & \vdots & \ddots & \vdots \\ 124751 & 124752 & \dots & 250 * 500 \end{bmatrix}.$$

Solution: even though everyone's times might be slightly different, in general appending columns/rows and using for loops is going to be more slow than taking advantage of numpy's functions. Python is a high-level programming language which is in general, easier to use than other faster languages such as C and C++. It's convenient and flexible (the amount of open source code implemented in Python is quite impressive and well documented). Yet Python is slow. Numpy provides libraries and functions for working with arrays (vectors, matrices, tensors) but they're backed by code written in high-speed low-level languages like C, C++, and Fortran. Because all of Numpy's operations take place outside Python, they're not constrained by this language's speed limitations. The following script assigns the matrices in different ways.

```
# Exercise I: Matrices in Python
import numpy as np
from time import perf_counter as tic
# We try different ways of assigning the matrix (these are some of the options)

size = (2,4)

# Manually entering the elements
t0 = tic()
M0 = np.empty(size)
c = 1
for i in range(size[0]):
    for j in range(size[1]):
        M0[i, j] = c
        c += 1
t1 = tic() - t0
print("Time taken entering manually each element: " + f"{t1:.2e}" + " s.")
```

```
# Filling the matrix row wise
t0 = tic()
M1 = np.empty(size)
for i in range(size[0]):
    M1[i, :] = np.arange(i*size[1] + 1, (i+1)*size[1] + 1)
t1 = tic() - t0
print("Time taken filling the matrix row wise: " + f"{t1:.2e}" + " s.")

# Filling the matrix column wise
t0 = tic()
M2 = np.empty(size)
for j in range(size[1]):
    M2[:, j] = np.arange( j+1, (size[0]-1)*size[1] + j + 2, size[1] )
t1 = tic() - t0
print("Time taken filling the matrix row wise: " + f"{t1:.2e}" + " s.")


# Using np arrange and np resize
t0 = tic()
M1 = np.resize(np.arange(1, 9), (4,2))
t1 = tic() - t0
print("Time taken using np.arange and np.resize: " + f"{t1:.2e}" + " s.")

# Using linspace
t0 = tic()
M2 = np.linspace( (1, 5), (4, 8), 4 ).T
t1 = tic() - t0
print("Time taken using np.linspace: " + f"{t1:.2e}" + " s.")
```

Exercise II Function in Python

We want to compute the function $f(x) = (\sqrt{1+x}-1)/x$ for different values of x in a neighborhood of 0. We first notice that $f(x)$ can be equivalently written as $f(x) = 1/(\sqrt{1+x}+1)$ and also as $f(x) = 1/2 - x/8 + x^2/16 - 5x^3/128 + o(x^4)$.
Create three function handles representing the above definitions of $f(x)$ (*hint*: the term $o(x^4)$ can be neglected in the computation) and, for each function handle,

a) evaluate $f(x)$ at $X = [10^{-10}\, 10^{-12}\, 10^{-14}\, 10^{-16}]$ using a `for` loop;

b) evaluate $f(x)$ at the same points given in a) using Python vector algebra;

c) display the results and comment on the importance of *round-off errors* for this example;

d) make sure you document your functions correctly;

e) if $x = 0$ then your first function raises an error explaining why.

Solution: the script below implements the solution of this question. Notice how the functions are commented.

```
# Exercise II: Function in Python
import numpy as np
from time import perf_counter as tic
```

```
x = np.array( [1e-10, 1e-12, 1e-14, 1e-16] )

# Function 1: direct

def f_direct(x):
    '''
    Implementation of the following function:
        ( sqrt(1 + x) - 1 )/x
    Parameters
    ----------
    x : fl, np.array
        Number or np array where to evaluate f(x)

    Raises
    ------
    ZeroDivisionError
        If x=0
    '''
    if np.any(x == 0):
        raise ZeroDivisionError("Can't divide by zero!")
    return (np.sqrt(1 + x) - 1)/x

def f_opt1(x):
    '''
    Implementation of the following function:
        ( sqrt(1 + x) - 1 )/x
    written as
        1/(sqrt(1 + x) + 1)
    Parameters
    ----------
    x : fl, np.array
        Number or np array where to evaluate f(x)
    '''
    return 1/(np.sqrt(1 + x) + 1)

def f_opt2(x):
    '''
    Implementation of the following function:
        ( sqrt(1 + x) - 1 )/x
    written as its O(x^4) approximation
       1/2 - x/8 + x^2/16 - 5*x^3/128
    Parameters
    ----------
    x : fl, np.array
        Number or np array where to evaluate f(x)
    '''
    return 0.5 - x/8 + x**2/16 - 5*x**3/128

print("\n\nEvaluating the functions using a for loop \n")
f_eval_for_0 = np.empty_like(x)
f_eval_for_1 = np.empty_like(x)
f_eval_for_2 = np.empty_like(x)
t0 = tic()
for i in range(len(x)):
    f_eval_for_0[i] = f_direct(x[i])
    f_eval_for_1[i] = f_opt1(x[i])
    f_eval_for_2[i] = f_opt2(x[i])
t1 = tic() - t0
```

```
print("Time taken for loop: " + f"{t1:.2e}" + " s.")

print("\n\nEvaluating the functions using numpy vector algebra \n")
t0 = tic()
f_eval_np_0 = f_direct(x)
f_eval_np_1 = f_opt1(x)
f_eval_np_2 = f_opt2(x)
t1 = tic() - t0
print("Time taken numpy vector algebra: " + f"{t1:.2e}" + " s.")

print("\n\nSolutions given by different implementations:")
print("Direct:   " + str(f_eval_np_0))
print("Method 1: " + str(f_eval_np_1))
print("Method 2: " + str(f_eval_np_2))
```

Exercise III Matrix-vector multiplication in Python

a) Consider the multiplication of a matrix $A \in \mathbb{R}^{m \times n}$ with a vector $v \in \mathbb{R}^n$. Write a Python file containing a script that:

- creates a matrix of dimension $m \times n$
- creates a vector of dimenstion n
- define a function that computes Av by using two nested loops

Solution: the script below implements the solution of this question.

```
 # Generated with ChatGPT
def matrix_vector_multiplication(matrix, vector):
    if len(matrix[0]) != len(vector):
        raise ValueError("Matrix and vector dimensions are not compatible for multiplication")

    result = [0] * len(matrix)

    for i in range(len(matrix)):
        for j in range(len(vector)):
            result[i] += matrix[i][j] * vector[j]

    return result

# Example usage:
matrix = [[1, 2, 3], [4, 5, 6]]
vector = [7, 8, 9]
result = matrix_vector_multiplication(matrix, vector)
print(result)
```

Exercise IV Matrix-vector multiplication with NumPy

a) Consider the same operation as in the previous exercice, the multiplication of a matrix $A \in \mathbb{R}^{m \times n}$ with a vector $v \in \mathbb{R}^n$. Compute matrix-vector multiplication by using numpy library:

- create a matrix of dimension $m \times n$, a vector of dimenstion n
- define a function that computes Av by using two nested loops
- compare the performance obtained for different values of m and n between the two nested loops code and the code using numpy library and draw a plot displaying the obtained performance

Exercise V Hello world with Python and MPI
Execute the following simple code on 4 processors several times.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print("I am rank = ", rank )
```

To execute this code, do

```
$ mpiexec -n 4 python script.py
```

Observe the order in which the prints take place.

Solution: you should notice that the ranks are printed in different random orders. They change every time you run this script.