Assignment 1 - Matrix-matrix multiplication

Contents

This assignment makes up 20% of the overall marks for the course. The deadline for submitting this assignment is **5pm on Thursday 20 October 2022**.

Coursework is to be submitted using the link on Moodle. You should submit a single pdf file containing your code, the output when you run your code, and your answers to any text questions included in the assessment. The easiest ways to create this file are:

- Write your code and answers in a Jupyter notebook, then select File -> Download as -> PDF via LaTeX (.pdf).
- Write your code and answers on Google Colab, then select File -> Print, and print it as a pdf.

Tasks you are required to carry out and questions you are required to answer are shown in bold below.

The assignment

In this assignment, we will look at computing the product AB of two matrices $A, B \in \mathbb{R}^{n\times n}$. The following snippet of code defines a function that computes the product of two matrices. As an example, the product of two 10 by 10 matrices is printed. The final line prints `matrix1 @ matrix2` - the `@` symbol denotes matrix multiplication, and Python will get Numpy to compute the product of two matrices. By looking at the output, it's possible to check that the two results are the same.

```
import numpy as np


def slow_matrix_product(mat1, mat2):
    """Multiply two matrices."""
    assert mat1.shape[1] == mat2.shape[0]
    result = []
    for c in range(mat2.shape[1]):
        column = []
        for r in range(mat1.shape[0]):
            value = 0
            for i in range(mat1.shape[1]):
                value += mat1[r, i] * mat2[i, c]
            column.append(value)
        result.append(column)
    return np.array(result).transpose()


matrix1 = np.random.rand(10, 10)
matrix2 = np.random.rand(10, 10)

print(slow_matrix_product(matrix1, matrix2))
print(matrix1 @ matrix2)
```

The function in this snippet isn't very good.

Part 1: a better function

Write your own function called `faster_matrix_product` that computes the product of two matrices more efficiently than `slow_matrix_product`. Your function may use functions from Numpy (eg `np.dot`) to complete part of its calculation, but your function should not use `np.dot` or `@` to compute the full matrix-matrix product.

Before you look at the performance of your function, you should check that it is computing the correct results. **Write a Python script using an `assert` statement that checks that your function gives the same result as using `@` for random 2 by 2, 3 by 3, 4 by 4, and 5 by 5 matrices.**

In a text box, **give two brief reasons (1-2 sentences for each) why your function is better than `slow_matrix_product`.** At least one of your reasons should be related to the time you expect the two functions to take.

Next, we want to compare the speed of `slow_matrix_product` and `faster_matrix_product`. **Write a Python script that runs the two functions for matrices of a range of sizes, and use `matplotlib` to create a plot showing the time taken for different sized matrices for both functions.** You should be able to run the functions for matrices of size up to around 1000 by 1000 (but if you're using an older/slower computer, you may decide to decrease the maximums slightly). You do not need to run your functions for every size between your minimum and maximum, but should choose a set of 10-15 values that will give you an informative plot.

Part 2: speeding it up with Numba

In the second part of this assignment, you're going to use Numba to speed up your function.

Create a copy of your function `faster_matrix_product` that is just-in-time (JIT) compiled using Numba. To demonstrate the speed improvement acheived by using Numba, **make a plot (similar to that you made in the first part) that shows the times taken to multiply matrices using `faster_matrix_product`, `faster_matrix_product` with Numba JIT compilation, and Numpy (`@`).** Numpy's matrix-matrix multiplication is highly optimised, so you should not expect to be as fast is it.

You may be able to achieve further speed up of your function by adjusting the memory layout used. The function `np.asfortanarray` will make a copy of an array that uses Fortran-style ordering, for example:

```
import numpy as np

a = np.random.rand(10, 10)
fortran_a = np.asfortranarray(a)
```

Make a plot that compares the times taken by your JIT compiled function when the inputs have different combinations of C-style and Fortran-style ordering (ie the plot should have lines for when both inputs are C-style, when the first is C-style and the second is Fortran-style, and so on). Focusing on the fact that it is more efficient to access memory that is close to previous accesses, **comment (in 1-2 sentences) on why one of these orderings appears to be fastest that the others**. (Numba can do a lot of different things when compiling code, so depending on your function there may or may not be a large difference: if there is little change in speeds for your function, you can comment on which ordering you might expect to be faster and why, but conclude that Numba is doing something more advanced.)

By Timo Betcke & Matthew Scroggs