Report 1: Matrix computation warmup and generation of Chebyshev polynomials

Don Winter[1] **and Madhulan Suresh**[2]

[1]don.winter@rwth-aachen.de, Mat. No. 431380
[2]madhulan.suresh@rwth-aachen.de, Mat. No. 429481

Introduction

The purpose of the first exercise is to get familiar with computations involving matrices and to get a first taste of implementing functions in programs from their mathematical definition. For this end we chose the Python programming language in conjunction with the popular NumPy[1] package which provides an efficient and convenient implementation of high dimensional arrays and functions to manipulate these. The exercise was stated in two parts. In the first part a 6×6 matrix of random values had to be generated, which was subsequently querried for maximum values together with their indices. A dot product between a row vector of the largest elements per column and a column vector of largest elements per row had to be carried out. Finally, the initial matrix needed to be multiplied with another random 6×6 matrix in forward and reversed order. The second part's task was to implement a (vectorized) function to produce the Chebyshev polynomials of the first kind up to order N+1 for a given column vector of x-values. The problem statement gave both an analytical expression and a recurrence relation, for which we chose to implement the latter. Subsequently, the first four polynomials had to be plotted. For this task we made use of the well-known Matplotlib visualization package[2].

Simulation model and method

In order to create a reproducable simulation we set the seed of NumPy's (pseudo-)random number generator (RNG) to a constant value in the first part of the exercise. Thereby, the subsequent creation of random matrices will call the RNG in a well-defined fashion, s.t. the random numbers generated for each matrix element are always the same each time we run the program. One way to create uniformly-random matrices in NumPy is by using the `numpy.random.uniform` function together with the desired range and the `size` argument to specify the number of elements per axis. To find maximum values in a NumPy array `numpy.max` can be used in conjuction with the `axis` argument, which allows to easily choose between row-wise (`axis=0`), column-wise (`axis=1`) or overall (`axis=(0,1)`) maximum values. Now, to find the (2-dimensional) index of the maximum value we can pattern-match the matrix again this value by `A==a_max`, which creates a boolean array of the same dimensions as A, which is `False` everywhere except at the positions where a matrix element fullfills the condition. Passing this *boolean mask* to the `numpy.where` function then extracts lists of row and column indices where the mask is `True`. We processed the resulting lists further to produce a list of (row, column) index tuples for better legibility. Furthermore, we needed to reshape one of the maximum value row vectors into a column vector in order to calculate the dot product. For this we used NumPy's indexing syntax `[:,numpy.newaxis]`, which inserts a new axis of one element in the column dimension, while the elements of the array are distributed along the row dimension. This gives a column vector. Another NumPy-specific syntax is the @ alias, which is equivalent to calling `numpy.matmul` on two matrices. For the second part we chose to implement the recurrence relation

$$T_n(x) = \begin{cases} 1 & \text{n} = 0 \\ x & \text{n} = 1 \\ 2xT_{n-1}(x) - T_{n-2}(x) & \text{n} > 1 \end{cases} \tag{1}$$

as a recursive function `T(x,n)` wrapped inside a function `cheby(x,N)` which produces a matrix of size `len(x)` $\times$ `(N+1)`. This wrapping makes `T` only available within the context of `cheby` which, although not strictly necessary, reduces statefullness of the code. This is commonly referred to as a *closure*. In order to generate the output matrix we first initiate an array of the desired shape to all zeros, over which we iterate column-wise by inserting T-values for a vector of x-values and an n up to N+1.

[1]https://numpy.org
[2]https://matplotlib.org/

Inside the T-function we make implicit use of NumPy's ability to perform arithmetic operations on vectors and to broadcast scalars and matrices to common dimensions, if unambiguous. Subsequently, we take the generated output matrix and iterate over the columns which we then plot against the x-values.

Simulation results

For the first part of the exercise there is not much to mention. Our matrix A has the form:

```
[[ 2.10687867 -1.30277895 -4.16588727  0.23133095  3.02053762 -0.62937121]
 [-1.22475236  3.64587694 -1.66830207 -3.24711163 -4.08132847  3.37226603]
 [-4.84911121  1.29181105  4.7473757   3.20814451 -0.41747964  0.32897331]
 [-1.93237674  2.13019924  2.89543416  2.0059373  -1.5993809  -3.37974114]
 [ 0.04742705 -1.90305693 -1.69189845  2.1386011  -3.59186815 -4.19460659]
 [-2.42373321 -0.57714717 -3.30048289 -3.58769653  3.16368819 -1.40570695]]
```

which has the largest value ~ 4.747 at index (2,2). The row vector of maximum column values is:

```
[2.10687867 3.64587694 4.7473757  3.20814451 3.16368819 3.37226603]
```

and the column vector of maximum row values:

```
[[3.02053762]
 [3.64587694]
 [4.7473757 ]
 [2.89543416]
 [2.1386011 ]
 [3.16368819]]
```

for which the dot product gives ~ 68.918. Concerning the product of two matrices it is worth mentioning that the C and D matrix are not equal, since the matrix multiplication is non-commutative. Or in other words it is highly improbable that two random matrices A and B will commute. We decided to not show this result here for brevity. The reader may reproduce our findings by executing the code in the Appendix.

For the second part we recognize by visual inspection that the resulting plot (Fig. 1) has the form of the familiar Chebyshev polynomials. Our results are pretty much identical to other resources[3].

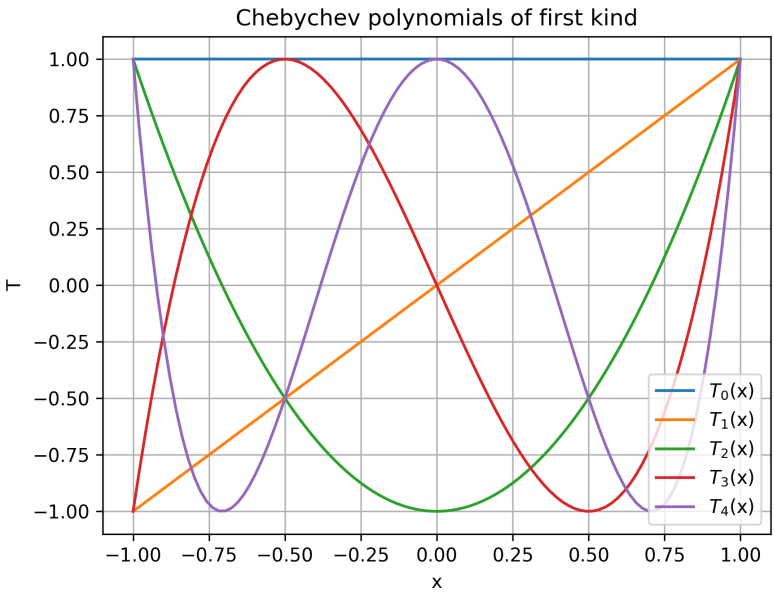

Figure 1. The first five T_n Chebyshev polynomials of the first kind

[3] https://en.wikipedia.org/wiki/Chebyshev_polynomials

Discussion

In conclusion, we explored various NumPy functions in order to efficiently create and carry out operations involving matrices. We learned how to craft reproducable simulations involving randomly generated numbers by setting the `seed` of NumPy's RNG from the `numpy.random` submodule. Finally, we practiced implementing a vectorized recursive function for which we plotted the result. The result is comparable to other sources.

Appendix

Part 1

```python
import numpy as np

np.random.seed(1380) # Set seed
A = np.random.uniform(-5, 5, size=(6,6)) # Create 6x6 random array, uniformly drawn from [-5,5]

a_max = np.max(A, axis=(0,1)) # Search along column and row axis for largest element
a_max_index = list(zip(*np.where( A == a_max ))) # Find indices in boolean mask

row = np.max(A, axis=0) # A row vector of largest elements per column
col = np.max(A, axis=1)[:, np.newaxis] # A column vector of largest elements per row
print(row @ col) # Matrix vector multiplication (dot product)

B = np.random.uniform(-5, 5, size=(6,6))
C = A @ B
D = B @ A
print(np.any(C == D))
```

Part 2

```python
import numpy as np
import matplotlib.pyplot as plt

def cheby(x: np.ndarray, N: int) -> np.ndarray: # declare input/output types
    """Calculate T values up to (inclusive) N for values in x."""

    assert np.all(x >= -1) and np.all(x <= 1) and N >= 0 # assert if arguments in range

    def T(x,n): # We only need T(x,n) to be available in the context of cheby. This justifies a closure.
        """Recursively calculate T values."""

        # Guard clauses as defined in the exercise.
        if n == 0:
            return 1
        elif n == 1:
            return x
        else:
            return 2 * x * T(x, n-1) - T(x, n-2) # recursive function call

    res = np.zeros((len(x),N+1)) # create array of zeros of desired shape
    for n in range(N+1): # iterate over all n values up to N+1
        res[:,n:n+1] = T(x,n) # fill columns with T vector corresponding to one n.

    return res

x = np.linspace(-1,1,100)[:,None] # column vector of test values
N = 4 # inclusive max. order for which to generate Chebychev polynomials

res = cheby(x,N) # generate matrix of shape len(x)*(N+1) matrix to store T values

# plot the result
for n,col in zip(range(N+1),res.T):
    plt.plot(x,col,label=r'$T_{%d}$(x)'%n)

plt.grid()
plt.title("Chebychev polynomials of first kind")
plt.xlabel("x")
plt.ylabel("T")
plt.legend()
plt.savefig("cheby.png", dpi=300)
```