3 Numerical Analysis I: Representations, Input and Output, and Graphics

Problem Solutions

A5

Problem 3.1 J3 Write a program that meets the following requirements:

a. It constructs a NumPy matrix `A` to represent the following mathematical matrix:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{bmatrix}.$$

b. It defines a function `left_up_sum(A: np.ndarray) -> np.ndarray` that adds the component (element) to the left and the component above (wrapping, if necessary) to each component. The function should pass through the array once, row-by-row, and return a new array. The function should be able to handle any size of matrix.

c. It defines a function `left_up_sums(A: np.ndarray, n: int) -> np.ndarray` that executes `left_up_sum()` `n` times and returns a new array.

d. It calls `left_up_sums()` on `A` and prints the returned array for the following values of `n`: `0`, `1`, `4`.

Solution 3.1 J3 The following program meets the requirements:

```
"""Solution to Chapter 3 problem J3"""
import numpy as np

# %% [markdown]
## Introduction
# This program will define two functions, left_up_sum() and
# left_up_sums(), and it will call left_up_sums() on a given
# matrix A for various numbers of sums.
# %% [markdown]
```

```
## Function Definitions
# %%
def get_element_wrapped(A: np.array, row: int, col: int) -> complex:
    """Returns an element from matrix with wrapped indices"""
    n, m = A.shape
    r = row % n  # Mod operator % returns remainder of division
    c = col % m
    return A[r, c]


def left_up_sum(A: np.ndarray) -> np.ndarray:
    """Adds the element to the left and the element above
    (wrapping, if necessary) to each component. Passes through
    the array once, row-by-row, and returns a new array.
    """
    shape = A.shape
    B = A.copy()  # Initialize output array
    for r in range(0, shape[0]):  # Each row
        for c in range(0, shape[1]):  # Each column
            left = get_element_wrapped(B, r - 1, c)
            up = get_element_wrapped(B, r, c + 1)
            B[r, c] = B[r, c] + left + up
    return B


def left_up_sums(A: np.ndarray, n: int) -> np.ndarray:
    """Calls left_up_sum() n times on A and returns the result"""
    for iteration in range(0, n):
        A = left_up_sum(A)
    return A


# %% [markdown]
## Call Function and Print
# %%
A = np.array(
    [
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    ]
)
test_sum_numbers = [0, 1, 4]

for n in test_sum_numbers:
    print(f"left_up_sums() of A for {n} sums:")
    print(left_up_sums(A, n))
```

This program prints the following to the console:

```
left_up_sums() of A for 0 sums:
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
left_up_sums() of A for 1 sums:
[[11 14 17 20 23 26 29 32 35 39]
 [32 37 42 47 52 57 62 67 72 90]]
left_up_sums() of A for 4 sums:
[[1069 1240 1411 1582 1753 1924 2096 2327 2829 3673]
 [2202 2543 2884 3225 3566 3907 4264 4811 5944 7640]]
```

Problem 3.2 ZF The inner product of two real n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=0}^{n-1} x_i y_i.$$

The result is a scalar. The `np.inner()` and `np.dot()` functions can be used in NumPy to find the inner product of two vectors of the same size. In this problem, we will write our own function that computes the real inner product even if they are of different sizes. Write a program that meets the following requirements:

a. It defines a function

```
inner_flat_trunc(x: np.ndarray, y: np.ndarray) -> float
```

that returns the truncated inner product of vectors `a` and `b` even if the sizes of the vectors do not match by using a truncated version of the one that is too long. The function should handle any shape of input arrays by using the `flatten()` method before truncating and taking the inner product. If both input arrays do not have `dtype` attribute `np.dtype('float')` or `np.dtype('int')`, the function should raise a `TypeError` exception.

b. It calls `inner_flat_trunc()` on the following arrays:
   i. A pair of arrays from the lists: `[-1.1, 3, 2.9, -1, -9.2, 0.1]` and `[1.3, 0.2, 8.3]`
   ii. An array of the integers from 0 through 13 and an array of the integers from 3 through 12
   iii. An array of 21 linearly spaced elements from 0 through 10 and an array of 11 linearly spaced elements from 5 through 25.
   iv. A pair of arrays of elements from the lists `[True, False, True]` and `[0, 1, 2]` (handle the exception in the main script so it runs without raising the exception)

Solution 3.2 ZF The following program meets the requirements:

```python
"""Solution to Chapter 3 problem ZF"""
import numpy as np

# %% [markdown]
## Introduction
# This program defines a function inner_flat_trunc() for computing
# the real inner product of two vectors that needn't be the same length
# %% [markdown]
## Function Definitions
# %%
def inner_flat_trunc(x: np.ndarray, y: np.ndarray) -> float:
    """Returns the real inner product of two vectors of potentially
    different lengths
    """
    # Check input dtypes
    ok_dtypes = [np.dtype("float"), np.dtype("int")]
    if not (x.dtype in ok_dtypes and y.dtype in ok_dtypes):
        raise TypeError("Both vectors must be of type float or int.")
    x = x.flatten()
    y = y.flatten()
    min_len = min(len(x), len(y))
    prod = 0  # Initialize inner product
    for i in range(min_len):
        prod += x[i] * y[i]
    return prod


# %% [markdown]
## Call Function and Print
# %%
test_vector_pairs = (
    (
        np.array([-1.1, 3, 2.9, -1, -9.2, 0.1]),
        np.array([1.3, 0.2, 8.3]),
    ),
    (np.arange(0, 14), np.arange(3, 13)),
    (np.linspace(0, 10, 21), np.linspace(5, 25, 11)),
    (np.array([True, False, True]), np.array([0, 1, 2])),
)

for vector_pair in test_vector_pairs:
    print(
        f"Product of\n\tx = {vector_pair[0]} and"
        f"\n\ty = {vector_pair[1]}:"
    )
    try:
        product = inner_flat_trunc(vector_pair[0], vector_pair[1])
```

```
    except TypeError as e:
        print(f"\t\tCaught TypeError: {e}")
        break
    print("\t\t", product)
```

This program prints the following to the console:

```
Product of
  x = [-1.1  3.   2.9 -1.  -9.2  0.1] and
  y = [1.3 0.2 8.3]:
     23.240000000000002
Product of
  x = [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13] and
  y = [ 3  4  5  6  7  8  9 10 11 12]:
     420
Product of
  x = [ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.
   ↪  6.5
  7.   7.5  8.   8.5  9.   9.5 10. ] and
  y = [ 5.  7.  9. 11. 13. 15. 17. 19. 21. 23. 25.]:
     522.5
Product of
  x = [ True False  True] and
  y = [0 1 2]:
    Caught TypeError: Both vectors must be of type float or int.
```

Problem 3.3 3H Consider the following mathematical matrices and vectors:

$$A = \begin{bmatrix} 2 & 1 & 9 & 0 \\ 0 & -1 & -2 & 3 \\ -3 & 0 & 8 & -4 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 9 & -1 \\ 1 & 0 & 3 \\ 0 & -1 & 1 \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad \boldsymbol{y} = \begin{bmatrix} 3 & 0 & -1 \end{bmatrix}. \tag{3.1}$$

Write a program that meets the following requirements:

a. It defines NumPy arrays to represent A, B, (column vector) $\boldsymbol{x}$, and (row vector) $\boldsymbol{y}$.
b. It computes and prints the following quantities:
   i. BA
   ii. $A^\top B - 6J_3$, where J_3 is the 3×3 matrix of all 1 components
   iii. $B\boldsymbol{x} + \boldsymbol{y}^\top$
   iv. $\boldsymbol{x}\boldsymbol{y} + B$
   v. $\boldsymbol{y}\boldsymbol{x}$
   vi. $\boldsymbol{y}B^{-1}\boldsymbol{x}$
   vii. CB, where C is the 3×3 submatrix of the first three columns of A