General instructions

- You have 2 hours to complete the examination. When applicable, people with special facilities have 2h20 minutes in total.
- The exam is "closed book", meaning that you can only make use of the material given to you.
- You are supposed to write the codes in the Python programming language, but no points are subtracted for syntax errors if is correct otherwise.
- **Do not use while statements.**
- **Do not use global variables.**
- **Elementary operations:** any mathematical operation on a scalar value such as multiplications, additions, comparisons (like max), absolute value, square root, power. Do not count accessing array elements in complexity calculations.
- Note that you could also try to compute the complexity of the algorithms without the coding part, in case you get stuck in the code. You need in any case to write down step by step the complexity calculations, the coefficient in front of the "n-term" is also expected in your answer.
- **From NumPy, you are allowed to use only these methods:** `shape`, `zeros`.
- **Do not use SciPy.**
- The grade is computed with the formula: points/totalpoints $\times 9 + 1$.
- Keep the names of the variables and functions as stated in the question, if applicable.
- **If you do not follow these instructions you will not receive any points in the respective question.**
- **PLEASE ANSWER EACH OF THE QUESTIONS ON SEPARATE PAPER SHEETS, SINCE THEY WILL BE GRADED SEPARATELY.** Hence, if you just use 1 or 2 pages for a question, you can split the answers by tearing the answer sheets. In case you decide not to write the answer for a specific question, also hand a sheet stating that.

Identities

$$\sum_{i=1}^{n} i = n\frac{n+1}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}, \quad \sum_{i=1}^{n} i^3 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}, \quad \sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a} \quad (a \neq 1)$$

Questions

In all questions, Python functions must be written that operate on tridiagonal matrices. Assume that these tridiagonal matrices are passed as *full* matrices (NumPy 2D arrays) as arguments to each function, i.e., including the zero entries. The algorithms implemented by the functions must take advantage of the sparsity structure of the tridiagonal matrices and operate only on the non-zero elements, hence no need of extracting the (outer) diagonals before operating. Checking if a matrix entry is zero with conditional statements is not necessary.

1. (3 points) Write a function that, given a non-symmetric tridiagonal matrix $B \in \mathbb{R}^{n\times n}$ as its argument, returns both the max norm $\|B\|_{\max}$ and the Frobenius norm $\|B\|_F$. You are allowed only to use the a "max function" among arrays of length three, at most. Compute the computational complexity in terms of n.

   The norms are defined, for a_{ij} the element in row i and column j of an arbitrary (full) matrix A, as

   $$\|A\|_{\max} = \max_{1\leq i,j\leq n} |a_{ij}|, \quad \|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}.$$

   Hint: Python has built-in functions `max(scalar1, ...)` and `abs(scalar)` that you can use here.

2. (3.5 points) Write a function that, given a non-symmetric tridiagonal matrix $B \in \mathbb{R}^{n\times n}$ and a NumPy array $\vec{x} \in \mathbb{R}^n$ as its arguments, returns the result of $B^T B\vec{x}$. You are neither allowed to use any transpose operator, nor to compute the matrix $B^T B$ explicitly. Compute the computational complexity in terms of n.

3. (1.5 points) This question deals with the computation of the determinant of a tridiagonal matrix.

   a) Modify the following function such that it takes into account the sparsity structure of the tridiagonal matrix $A \in \mathbb{R}^{n\times n}$ and does not operate on the known zero entries of the matrix. Assume the function `subblock(A,i,j)` given, which returns a matrix of size $\mathbb{R}^{(n-1)\times(n-1)}$ computed by deleting the i-th row and j-th column of an arbitrary matrix (no need to be tridiagonal).

```python
def determinant(A):
    n = A.shape[0]
    if n == 1:
        # base case: stop the recursion
        det = A
    else:
        # recursion
        det = 0
        for i in range(n):
            B = subblock(A, 0, j)
            det += (-1)**i * A[0, i] * determinant(B)
    return det
```

   b) Compute the computational complexity in terms of n (do not include the cost of `subblock`) of the sparse tridiagonal version of the algorithm in a). Note that the complexity is not polynomial in terms of n.