Objective

The goal of this exercise is to design and implement a C++ framework for basic matrix operations. The framework must rely on inheritance, polymorphism, and templates, and expose its functionality to Python using pybind11.

Background

Let $A, B, C \in \mathbb{R}^{m \times n}$ be matrices and $k \in \mathbb{R}$ a scalar. The following operations are defined:

- **Matrix addition**: $C_{ij} = A_{ij} + B_{ij}$
- **Scalar multiplication**: $B_{ij} = k \cdot A_{ij}$
- **Trace**: $\mathrm{Tr}(A) = \sum_i A_{ii}$

All operations are element-wise except for the trace.

Instructions

You are asked to build a reusable matrix operations library in C++. The library must allow different operations to be applied to matrices through a common interface and must support different numeric types using templates.

Tasks

1. **(1 point) Define a templated `Matrix` class**

   Define a templated class `Matrix<T>` that:

   - Stores a 2D matrix of elements of type `T`.
   - Provides constructors for specifying matrix sizes.
   - Provides methods to query the number of rows and columns.

2. **(2 points) Element access and validation**

   - Implement a clearly defined interface (e.g., through operator overloading) for element access in the `Matrix<T>` class.
   - Ensure const-correctness.
   - Accessing invalid indices must throw exceptions.

3. **(2 points) Matrix operation hierarchy**

   - Define an abstract base class `MatrixOp<T>` with a pure virtual method:

     ```
     virtual Matrix<T> apply(const Matrix<T>& A) const = 0;
     ```

   - Implement the following derived classes:
     - `AddOp<T>`: Performs matrix addition with a second matrix.
     - `ScalarMultOp<T>`: Performs scalar multiplication.
   - Each derived class must:
     - Override `apply()`.
     - Validate dimensions where needed.
     - Throw exceptions on invalid operations.

4. **(2 points) Trace operation**

   - Implement a `TraceOp<T>` class, derived from `MatrixOp<T>`, that computes the trace of a matrix.
   - The operation must only be valid for square matrices (handle non-square matrices with exceptions).

- Clearly define how the trace result (which is a scalar) is returned.

5. **(1 point) Testing**

   In `main()`, demonstrate the use of polymorphism by applying the different matrix operations through a base-class reference or pointer. Verify also that the framework works for different numeric types.

6. **(Bonus, 2 points) Integration with Eigen**

   Integrate the Eigen linear algebra library and compare its results and performance with your implementation.

7. **(2 points) Configuration and compilation**

   - Write a `CMakeLists.txt` to compile your code into a library and a test executable.
   - Include proper C++17 standard requirements and optimization flags.
   - Provide clear build and usage instructions.

8. **(5 points) Python bindings using pybind11**

   - Expose the `Matrix` class and all matrix operations to Python.
   - Ensure that the Python bindings handle exceptions properly.
   - Write a Python script that:
     - Creates matrices.
     - Applies matrix operations.
     - Compares results and performance obtained through the Python bindings with:
       * The native C++ implementation.
       * NumPy or other packages.

Evaluation criteria

- Correct use of inheritance, polymorphism, and templates.
- Correctness of matrix operations.
- Proper handling of edge cases.
- Memory and exception safety (proper RAII, no memory leaks).
- Correct and usable Python bindings, with a clean, Pythonic interface.
- Meaningful comparison with established numerical libraries (NumPy, SciPy).
- Quality of the CMake configuration.
- Code clarity and organization.