Exercise session 12

Integrating C++ and Python codes.

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

10 Dec 2024

Exercise 1: binding classes and magic methods

Provide Python bindings using pybind11 for the code provided as the solution to exercise 1 from session 03.

1. Bind the `DataProcessor` class and its member functions. Using a lambda function, expose a constructor taking a Python list as an input, to be converted to a `std::vector` and invoking the actual constructor.
2. Provide Python bindings for the addition (`__add__`), the read (`__getitem__`) and write (`__setitem__`) access, and the output stream (`__str__`) operators.
3. Package the Python module with the compiled C++ library using `setuptools`.
4. Write a Python script to replicate the functionalities implemented in the `main.cpp` file.

Exercise 2: binding class templates and exceptions

Provide Python bindings using pybind11 for the code provided as the solution to exercise 2 from session 05.

1. Modify the `NewtonSolver::solve()` method in order to throw a `std::runtime_error` exception instead of returning `NaN` when failed to converge to a root.
2. Bind the `NewtonClass` class and its member functions, providing explicit instantiations for `double` and `std::complex<double>` numbers. The Python interface should provide consistent default arguments. Python bindings should be implemented in a separate `newton_py.cpp` file. Translate the `std::runtime_error` C++ exception to a `RuntimeError` Python exception.
3. Use CMake to setup the build process.
4. Write a Python script to replicate the functionalities implemented in the `main.cpp` file.
5. Verify that exception handling works properly.

Exercise 3: binding with external libraries

1. Implement C++ functions using the Eigen library to perform matrix-matrix multiplication and matrix inversion.
2. Provide Python bindings using pybind11 for the code implemented.
3. Use CMake and `setuptools` to setup the build process.
4. Write a Python script to test the performance of the Eigen-based operations. Implement a `log_execution_time` decorator to print the execution time of a function.
5. Compare the execution time of these operations to equivalent operations in NumPy (e.g., `numpy.matmul` for multiplication and `numpy.linalg.inv` for inversion). Use a large matrix (e.g., 1000×1000) of random integers between 0 and 1000 for the test.