Exercise session 12

Integrating C++ and Python codes.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

10 Dec 2025

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 that takes a Python list as input, converts it to a `std::vector`, and invokes the actual C++ constructor.
2. Provide Python bindings for:
    - Addition operator (`__add__`)
    - Read/write access operator (`__getitem__`, `__setitem__`)
    - Output stream operator (`__str__`)
3. Package the Python module with the compiled C++ library using `setuptools`.
4. Write a Python script to replicate the functionalities from 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 3 from session 05.

1. Modify the `NewtonSolver::solve()` method to throw a `std::runtime_error` exception instead of returning `NaN` when it fails to converge to a root.
2. Bind the `NewtonSolver` class and its member functions, providing explicit instantiations for `double` and `std::complex<double>` types. The Python interface should provide consistent default arguments. Translate the `std::runtime_error` C++ exception to a `RuntimeError` Python exception. Implement Python bindings in a separate `newton_py.cpp` file.
3. Use CMake to set up the build process.
4. Write a Python script to replicate the functionalities implemented in the `main.cpp` file, and 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 implemented code.
3. Use CMake and `setuptools` with `pyproject.toml` to set up 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 each function.
5. Compare the execution time of these operations to equivalent NumPy operations (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.