Homework 02

Implementation of a Scientific Computing Toolbox

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa, Giuseppe Alessio D'Inverno

Due date: 12 Dec 2024

Objective

Develop a toolbox in C++ covering various data science and scientific computing areas.
Implement **module A)** and **one module among B), C), D)**, of your choice.

Components

A) **Statistics module** ⚠️ **(mandatory)**

B) **Interpolation module**

C) **Numerical integration module**

D) **ODE module**

Requirements and descriptions for each module are provided in the sections below.

A) Statistics module ⚠️ (mandatory)

Implement a module to perform statistical analyses on data imported from a `CSV` or a `JSON` file and output relevant information to a text file. Parse input and output filenames as command line arguments.

- Implement utilities for statistical operations like mean, median, standard deviation, variance, frequency count, classification, and correlation analyses.
- Implement iterators for seamless data traversal.
- Consider using `std::variant` for storing numerical or categorical data and `std::optional` for possibly missing/NA values.
- Select a dataset from Kaggle, and use it to test your implementation by performing statistical analysis on a real application.

B) Interpolation module

Implement a module to support **composite** linear and polynomial interpolation, and (**bonus**) cardinal cubic B-spline interpolation of a given set of data $\{(x_i, y_i)\} \subset \mathbb{R}^2$.

- Implement a common interface that stores a list of nodes $\{x_i\}$ over an interval $[a, b]$ and the corresponding observed values $\{y_i\}$ for all kinds of interpolation.
- The implemented class(es) should expose a call operator (`operator()`) returning the interpolated value at a given point.
- Test your implementation through practical examples. Showcase the accuracy, efficiency, and order of convergence of each method implemented.

C) Numerical integration module

Implement a module for approximating integrals using **composite** numerical integration formulas of the form

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i),$$

where $\{w_i\}$ and $\{x_i\}$ are the weights and nodes of the quadrature formula, respectively.

- Consider methods such as the midpoint rule, the trapezoidal rule, Simpson's rule, and (**bonus**) Gaussian quadrature formulas.
- Test your implementation through practical examples. Showcase the accuracy, efficiency, and order of convergence of each method implemented.

D) ODE module

Implement a module for solving Ordinary Differential Equations (ODEs) of the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}),$$

where $\mathbf{y} \in \mathbb{R}^N, \mathbf{f} : \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N$, using explicit Runge-Kutta methods, such as Forward Euler, RK4, and (**bonus**) the explicit midpoint method.

- The module should handle both scalar ($N = 1$) and vector ODE problems.
- Output the solution to a `csv` file with header columns `t, y1, y2, ..., yN`.
- Test your implementation through practical examples. Showcase the accuracy, stability, efficiency, and order of convergence of each method implemented.

General guidelines

1. Emphasize the use of **modern C++** features, including STL containers, algorithms, iterators, smart pointers, and other utilities.
2. Utilize either *run-time* (class abstraction and inheritance) or *compile-time* (templates and policies) polymorphism, **providing motivation for your choice**.
3. Write error-safe code and **handle exceptions properly**.
4. Provide **clear documentation** of code design, algorithms, and decisions made.
5. Promote **code readability**, **modular design**, and adherence to **coding standards**.
6. Provide **sample applications** demonstrating the functionality of each module.

Integration of third-party libraries

- The integration of third-party libraries is highly encouraged, such as:
  - `Boost` (e.g., the modules `Histogram`, `JSON`, `Math`, `Odeint`).
  - `Eigen`, for linear algebra classes (vectors, matrices, linear solvers).
  - `GetPot`, for parsing comand line arguments and configuration files.
  - `GNU GSL`, for a wide range of mathematical routines.
  - `muParserX`, to parse string expressions such as `"sin(pi * x) * exp(x)"` as mathematical functions.

  or any other library of your choice, and showcase their synergy with your code.
- Discuss considerations and challenges in using third-party libraries.

Code organization

- Organize your implementation into subfolders and files with meaningful names.
- Ensure a clear separation between function declarations and definitions by placing them in different files whenever possible.
- The **2** modules implemented should be part of the same framework, e.g., by sharing namespaces, styling, and common utilities. However, each of them should be **compilable as a standalone shared library**, allowing independent use.
- Before submission, ensure your code's compatibility with various compilers by testing it, e.g. on GodBolt, and enabling the following compilation flags:

```
-std=c++17 -Wall -Wextra -Wpendantic -Werror
```

Submission

1. Include a `README` file that:
    - Clearly states **which module(s)** you implemented.
    - Lists all **group members** with their name, email address, and highlights their **individual contribution** to the project.
    - Provides a concise discussion of the obtained results, such as insights from statistical analyses, observations on the convergence order of implemented methods, and any other pertinent information.
2. Provide a working compilation script as the preferred method for building the libraries and testing your implementation. Clearly specify the commands needed to compile the code successfully. **Bonus**: use `Makefile` or `CMake` as build tools.
3. Submit a **single** compressed file (named `Homework_02_Surname1_Surname2.ext`) containing all source code, the `README`, and any other relevant files or third-party libraries (please comply to their licences).

Evaluation grid

1. Module A) + Module B), C), or D) (up to **1 point** each):
   - Successful compilation, implementation correctness, results correctness.
2. Effective utilization of modern C++ features (up to **1 point**):
   - STL, smart pointers, exceptions, const correctness, ...
3. Documentation, build instructions, and discussion of results (up to **1 point**)
4. Code organization (up to **0.5 points**):
   - Clear separation between function declarations and definitions, consistent use of namespaces and styling, organized file structure with meaningful names, compilability as standalone shared libraries.
5. Integration of third-party libraries (up to **0.5 points**)
6. **Bonus** points (up to **1 point**).