Advanced Programming - Exam 07 Jul 2025 - Part 2

Objective

Implement a C++ program that performs **gradient descent optimization** for multivariate functions. The program should use **inheritance, polymorphism, and templates** to support various optimization strategies and floating-point types.

Mathematical description

Given a multivariate function $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^n$, the gradient descent method iteratively minimizes the function by updating the parameter vector:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)})$$

where:

- $\mathbf{x}^{(k)}$ is the parameter vector at iteration k
- $\alpha > 0$ is the **learning rate** (step size)
- $\nabla f(\mathbf{x}^{(k)})$ is the **gradient** of f evaluated at $\mathbf{x}^{(k)}$

The algorithm terminates when either:

- Maximum number of iterations is reached, or
- The gradient norm falls below a tolerance: $||\nabla f(\mathbf{x}^{(k)})|| < \epsilon$.

Exercise instructions

Overview

Your task is to build a gradient descent optimizer using C++ and apply object-oriented design principles:

- **Inheritance**: Create a base class `Optimizer` and derive specific implementations.
- **Polymorphism**: Use virtual methods to dispatch between optimization strategies.
- **Templates**: Make the optimizer work with different numeric types (`float`, `double`) and with different problem dimension `N`.

Tasks

1. **(1 point) Define a base class `Optimizer`**
    - Include a pure virtual function:

      ```
      virtual std::vector<double> minimize(
          std::function<double(const std::vector<double>&)> f,
          std::function<std::vector<double>(const std::vector<double>&)> grad_f,
          std::vector<double> initial_x,
          std::size_t max_iterations = 1000,
          double tolerance = 1e-6) = 0;
      ```
    - Include methods for setting learning rate and accessing optimization history (function values, solutions, iteration number).
2. **(2 points) Implement `GradientDescentOptimizer`**
    - Override `minimize()` using the standard gradient descent algorithm.
    - Store optimization history (function values, solutions, iteration number).
    - Implement proper convergence checking based on gradient norm.
    - Handle edge cases (empty gradients, invalid learning rates).
3. **(1 point) Use polymorphism**
    - In `main()`, define a pointer or reference to the base class `Optimizer` and dispatch to the concrete implementation at runtime.
4. **(2 points) Test your implementation**
    - Minimize the following functions with **exact gradients**:
        - **Quadratic function**: $f(x, y) = x^2 + y^2$ with exact gradient $\nabla f = (2x, 2y) \to$ Minimum: $(0, 0)$ with $f = 0$.
        - **Rosenbrock function**: $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ with exact gradient:
            * $\frac{\partial f}{\partial x} = -2(1 - x) - 400x(y - x^2)$,
            * $\frac{\partial f}{\partial y} = 200(y - x^2)$,
            * Minimum: $(1, 1)$ with $f = 0$.
    - Test convergence for different learning rates and starting points.
5. **(2 points) Templatize your code**
    - Create a new class template `GradientDescentOptimizerT`.
    - Allow the optimizer to work with both `float` and `double` types.
    - Template the optimizer on the problem dimension `N` to enable compile-time optimization for fixed-size problems.
    - Use `std::array<T, N>` for fixed-dimension problems and compare performance with `std::vector<T>`.
    - Test both versions (templated and non-templated) in the same program.
6. **(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.
7. **(5 points) Python bindings using pybind11**

- Expose your C++ optimizer class and methods to Python using `pybind11`.
- Allow users to pass Python functions (function and gradient) to be optimized.
- Provide access to optimization history and convergence information.
- Demonstrate the binding with a Python script that:
  - Optimizes the same test functions.
  - Plots the optimization trajectory using `matplotlib`.
  - Compares your implementation with `scipy.optimize.minimize`.

Evaluation criteria

- Code design and modularity (proper class hierarchy).
- Correct use of inheritance, polymorphism, and templates.
- Accuracy of the optimization results and convergence behavior.
- Memory and exception safety (proper RAII, no memory leaks).
- Clean Python interface with visualization capabilities.
- Meaningful comparison with established optimization libraries (SciPy).
- Build system quality.