ChatGPT 5.2

Thought for 20s

According to an uploaded homework document (Level 9: Introductory Computational Finance), the original assignment covers: **exact Black–Scholes prices + parity + "batches" for testing**, **Greeks + divided differences**, **perpetual American options**, **Monte Carlo (Euler–Maruyama) + stress tests + SD/SE**, **Excel visualization**, and a **finite-difference stability experiment**. Level 9 Homework

Level 9 Homework Level 9 Homework Level 9 Homework

Below is a **complete, standalone "Modern C++20" version** you can follow independently (not "replace X with Y"). It keeps the *same finance content and test targets* but forces modern C++ design.

Level 9 Modern C++20 Homework: Option Pricing Library

0) Rules and constraints (mandatory)

Language & build

- Use **C++20**.
- Build with **CMake**.
- Compile with warnings-as-errors: `-Wall -Wextra -Wpedantic -Werror`.

Design rules

1. **No "parameter bags" with public members.** Any domain object must be a `class` with invariants enforced in constructors.
2. **No functions that take 5–6 raw doubles.** Use **strong types** and **parameter objects**.
3. No raw `new/delete`. Use RAII only (`std::unique_ptr`, `std::vector`, stack objects).
4. Prefer `std::span`, `std::ranges`, and **algorithms** over manual loops when reasonable.
5. Normal CDF must be implemented via a **policy** (Boost-based vs std::erf-based) so you can swap it.

Testing (mandatory)

Write unit tests that verify **Batch 1–4** exact prices and put–call parity (tolerances allowed). The batches and expected values are given in the original spec. Level 9 Homework Level 9 Homework

---

1) Domain types (strongly typed parameters)

Create strong types (you can use your own template strong type, or Boost strong typedef—your call). You must have at least:

- `Spot`, `Strike`, `Vol`, `Rate`, `Carry`, `TimeToExpiry`

And a validated parameter class:

cpp Copy code

```cpp
class BlackScholesParams {
public:
  BlackScholesParams(Strike K, TimeToExpiry T, Rate r, Vol sigma, Carry b);

  Strike K() const noexcept;
  TimeToExpiry T() const noexcept;
  Rate r() const noexcept;
  Vol sigma() const noexcept;
  Carry b() const noexcept;

private:
  // store values privately; validate invariants:
```

```
    // K > 0, T >= 0, sigma >= 0, etc.
};
```

The original homework explicitly asks you to encapsulate parameters (it used a public struct), but your version must use a class with invariants. Level 9 Homework

---

2) Normal CDF policy (compile-time selectable)

Implement **two** CDF policies:

(A) Boost policy

Uses Boost.Math normal distribution CDF.

(B) STL policy

Uses `std::erf` with:

$$\Phi(x) = 0.5 \cdot \left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

Your pricer must be templated on the policy:

cpp Copy code

```cpp
template <class NormalCdfPolicy>
class BlackScholesPricer {
public:
  double call(Spot S, const BlackScholesParams& p) const;
  double put (Spot S, const BlackScholesParams& p) const;
};
```

---

3) Section A — Exact European option prices (Black–Scholes)

A.1 Implement exact call/put pricing

Implement the generalized Black–Scholes formula with carry `b` (and for stock options use `b = r` as noted). Level 9 Homework Level 9 Homework

Your code must compute:

- d_1, d_2
- Call C
- Put P

(You can follow the formulas provided in the document.) Level 9 Homework Level 9 Homework

A.0 Test datasets (Batches)

Hard-code these batches in tests and verify the expected prices:

- **Batch 1:** `T=0.25, K=65, sig=0.30, r=0.08, S=60 → C=2.13293, P=5.84584` Level 9 Homework
- **Batch 2:** `T=1.0, K=100, sig=0.2, r=0.0, S=100 → C=7.96632, P=7.96632` Level 9 Homework
- **Batch 3:** `T=1.0, K=10, sig=0.50, r=0.12, S=5 → C=0.204121, P=4.0733` Level 9 Homework
- **Batch 4:** `T=30.0, K=100, sig=0.30, r=0.08, S=100 → C=92.1749, P=1.24651` Level 9 Homework

A.1(b) Put–call parity utility

Implement put–call parity and test it against the direct put formula. The document gives parity as:

$$C + Ke^{-rT} = P + S$$

Level 9 Homework

Your deliverable:

- Function/class that computes the missing price given the other.
- Tests for Batches 1–4.

A.1(d/e) Curve generation (ranges + mesh)

You must generate option prices for a monotonically increasing range of **S** (e.g., 10, 11, ..., 50), storing in a vector. The original assignment explicitly mentions writing a **mesh array** function and printing output.
Level 9 Homework

In your modern version:

- Implement:

cpp Copy code

```cpp
std::vector<Spot> make_spot_grid(Spot start, Spot end, double h);
```

- Use `std::ranges::transform` to compute call/put vectors.
- Output as CSV (instead of Excel).

Also compute:

- Prices as a function of **expiry T** and **volatility sigma** (same idea as the original part e).
  Level 9 Homework

---

4) Section A.2 — Greeks + numerical differentiation

A.2(a) Exact Greeks

Implement at least:

- **Delta (call & put)**
- **Gamma (call & put)**

The doc lists example Greek formulas and a specific gamma test dataset:
`K=100, S=105, T=0.5, r=0.1, b=0, sig=0.36` and asks to implement gamma (and also mentions delta). Level 9 Homework Level 9 Homework

Also, the doc includes a delta test value:

- exact delta call `0.5946`, delta put `-0.3566` (for the dataset they reference in that section).
  Level 9 Homework

A.2(b) Delta curve over S

Compute delta(call) for S over a range (10..50 style), store results in a vector, using your mesh function.
Level 9 Homework

A.2(c) Divided differences (finite differences)

Implement 3-point second-order finite difference approximations for:

- first derivative wrt S (delta approximation)
- second derivative wrt S (gamma approximation)

The doc gives the exact stencils and asks you to compare accuracy for multiple values of `h`, warning about round-off/subtractive cancellation. Level 9 Homework

Your deliverable:

- A reusable templated finite-difference module:

cpp Copy code

```cpp
template <class F>
double first_central(F&& f, double x, double h);

template <class F>
double second_central(F&& f, double x, double h);
```

- A small experiment output (CSV/table): exact vs FD for several `h`.

---

5) Section B — Perpetual American options

Implement perpetual American **call** and **put** formulae (no expiry `T`). The original spec provides formulas and explicitly says "no T parameter." Level 9 Homework Level 9 Homework

B(a) Parameter class

Create:

cpp Copy code

```cpp
class PerpetualParams {
public:
  PerpetualParams(Strike K, Vol sigma, Rate r, Carry b);
  // invariants
};
```

B(b) Test value

Test using:

`K=100, sig=0.1, r=0.1, b=0.02, S=110`

Expected:

- `C=18.5035`
- `P=3.03106`

Level 9 Homework

B(c) Price curve over S

Compute perpetual call/put for S=10..50 and store in vectors (use the same mesh/grid utility).

Level 9 Homework

---

6) Section C — Monte Carlo (Euler–Maruyama)

The original homework's Monte Carlo section simulates the underlying via time discretization and random Gaussian increments (Boost Random mentioned), then discounts average payoff. Level 9 Homework

Level 9 Homework

C(a) Implement a simulation engine

Implement:

- RNG wrapper (seedable)
- `simulate_terminal_spot(...)`
- `price_call_mc(...)` (and put)

Use:

- `NT` time steps
- `NSIM` simulation paths

C(b) Accuracy experiments: Batches 1 and 2

Run MC with Batches 1 and 2 and experiment with `NT` and `NSIM` until you match exact pricing accuracy (the doc explicitly asks this). Level 9 Homework

C(c) Stress test: Batch 4 to 2 decimals

For Batch 4, find `NT` and `NSIM` required for accuracy to two decimals. Level 9 Homework

Deliverables:

- A report (markdown or printed output) listing the chosen `(NT, NSIM)` and achieved error.

---

7) Section D — Advanced Monte Carlo: SD and SE

Implement generic functions for **standard deviation (SD)** and **standard error (SE)** based on the formulas in the doc, taking:

- vector of simulated discounted payoffs (size `M = NSIM`)
- `r` and `T` (as needed)

The document gives SD/SE definitions and requires generic functions integrated into the MC test program. Level 9 Homework Level 9 Homework

Deliverables:

- Functions:
  - `stddev(span<const double> payoffs, Rate r, TimeToExpiry T)`
  - `stderr(span<const double> payoffs, Rate r, TimeToExpiry T)`
- Output SD/SE for Batches 1 and 2 across a few `(NT, NSIM)` pairs. Level 9 Homework

---

8) Section E — "Excel visualization" replaced with CSV

The original uses an Excel package and asks you to print curves to Excel. Level 9 Homework Level 9 Homework

In your modern version:

- Implement a small CSV writer and output:
  - price vs S
  - price vs T
  - price vs sigma
  - delta vs S
  - perpetual price vs S

(You can still open CSV in Excel.)

---

9) Section F — Finite difference stability experiment

The doc notes explicit Euler stability: time step `k` must be "much less than" space step `h` (worst-case like $k = O(h^2)$), and asks you to determine relationships and run the provided FD code against batches. Level 9 Homework

In your modern version, you must:

- Create a `FdmStabilityRunner` that sweeps:
  - multiple `h` values
  - increasing `k` until instability/inaccuracy is observed
- Log results to CSV.

Also incorporate the remark that for an explicit method there's a relationship like `N = J*J` (time points vs

space points), and try smaller values too. Level 9 Homework

---

10) Required project layout (submit this)

swift Copy code

```
pricing_lib/
  CMakeLists.txt
  include/
    strong_types.hpp
    black_scholes_params.hpp
    perpetual_params.hpp
    normal_cdf_policy.hpp
    black_scholes_pricer.hpp
    perpetual_pricer.hpp
    mesh.hpp
    finite_difference.hpp
    mc_pricer.hpp
    stats.hpp
    csv_writer.hpp
  src/
    black_scholes_pricer.cpp   (optional if header-only)
    perpetual_pricer.cpp
    mc_pricer.cpp
    stats.cpp
    csv_writer.cpp
  apps/
    run_exact.cpp      // prints batch checks + writes curves
    run_greeks.cpp     // exact vs FD, writes tables
    run_mc.cpp         // batch MC experiments + SD/SE
    run_fdm.cpp        // stability sweep
  tests/
    test_exact.cpp     // Batches 1–4 exact + parity
    test_perpetual.cpp // perpetual known values
    test_greeks.cpp    // delta/gamma exact vs FD sanity
```

If you want, I can also generate a **grading rubric** (what gets points) and "minimum passing implementation" checklist (so you don't overbuild).

Sources