PDE-based Pricing in C++

An explicit finite-difference solver for Black–Scholes with dividends

Programming Project Report

Sacha Faye **Mathys Gouin** **Samuel Macé**

Repository (public):
https://github.com/Samgit0532/Pricing_par_EDP

Abstract. We implement a modular C++ pricing engine for several derivative products by numerically solving the Black–Scholes–Merton PDE with continuous dividends using an explicit finite-difference scheme. The code supports European and American derivatives (via early-exercise projection), returns price and Greeks (Delta, Gamma), and includes a test suite validating basic financial identities and numerical sanity checks (put–call parity, forward closed-form, dominance of American options, and composite product consistency). A central engineering contribution is an automatic grid construction driven by a user-chosen spatial resolution parameter `rel_dS`, such that $\Delta S = \texttt{rel\_dS} \cdot S_0$, designed to preserve accuracy near the spot while keeping runtimes reasonable.

Contents

1 Project architecture (high-level overview)

Our codebase is organized as follows (rooted in `src/`):

```
src/
  main.cpp                 (interactive application)
  tests/
      TestPricing.cpp     (automated consistency tests)
  model/
      BlackScholesModel.hpp
  grid/
      FdGrid.hpp
      GridParameters.hpp
  products/
      InterfaceProducts.hpp
      EuropeanCall.hpp     EuropeanPut.hpp
      AmericanCall.hpp     AmericanPut.hpp
      Future.hpp
      BullCallSpread.hpp   BearPutSpread.hpp
      Straddle.hpp
  solvers/
       ExplicitFdSolver.hpp
       Solver.cpp          (implementation of ExplicitFdSolver::price)
```

1.1 How modules interact

The core pricing pipeline is always the same:

1. We define market parameters in `model/BlackScholesModel.hpp`.
2. We instantiate a product (payoff + boundary conditions) from `products/`.
3. We build a finite-difference grid using `grid/GridParameters.hpp`, which returns an `FdGrid` from `grid/FdGrid.hpp`.
4. We call `ExplicitFdSolver::price` (declared in `solvers/ExplicitFdSolver.hpp`, implemented in `solvers/Solver.cpp`).
5. The solver returns price + Greeks (Delta, Gamma) and the entire solution slice at $t = 0$.

1.2 What the program takes as inputs

At runtime (interactive application in `src/main.cpp`), the user provides:

- Market inputs: S_0, r, σ, q.
- Contract inputs: maturity T, strike(s) depending on product.
- Numerical input: a spatial resolution parameter `rel_dS` such that $\Delta S = \texttt{rel\_dS} \cdot S_0$.

The test program (`src/tests/TestPricing.cpp`) uses fixed parameter sets and verifies financial identities and numerical properties automatically.

2 Model: Black–Scholes–Merton with dividends

2.1 Risk-neutral dynamics

We assume the underlying price S_t follows (under the risk-neutral measure):

$$dS_t = (r - q)S_t\, dt + \sigma S_t\, dW_t,$$

where r is the risk-free rate, σ the volatility, and q the continuous dividend yield.

2.2 Pricing PDE

For a derivative value $V(t, S)$, the Black–Scholes–Merton PDE is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - q)S\frac{\partial V}{\partial S} - rV = 0.$$

The PDE is solved backward in time with:

- terminal condition (payoff): $V(T, S) = \Phi(S)$,
- boundary conditions as $S \to S_{\min}$ and $S \to S_{\max}$.

2.3 Implementation in our code

The model parameters are stored in `model/BlackScholesModel.hpp`:

$$(r, \sigma, q),$$

and used both:

- in `solvers/Solver.cpp` to build finite-difference coefficients (drift $r - q$, discount rate r),
- in product boundary conditions (e.g. call boundary uses $e^{-q(T-t)}$ and $e^{-r(T-t)}$).

3 Solver: explicit finite differences (`solvers/ExplicitFdSolver.hpp`, `solvers/Solver.cpp`)

3.1 Terminal condition

For each product, we initialize:

$$V_i^{N_t} = \Phi(S_i),$$

where Φ is the payoff, implemented as `option.payoff(S[i])`.

3.2 Explicit backward time stepping

Let $V_i^n \approx V(t_n, S_i)$. For each time step $n = N_t - 1, \dots, 0$, we compute for interior nodes $i = 1, \dots, N_s - 1$:

$$V_i^n = A_i V_{i-1}^{n+1} + B_i V_i^{n+1} + C_i V_{i+1}^{n+1},$$

with:

$$A_i = \frac{\Delta t}{2}\left(\frac{\sigma^2 S_i^2}{\Delta S^2} - \frac{(r-q)S_i}{\Delta S}\right),$$
$$B_i = 1 - \Delta t\left(\frac{\sigma^2 S_i^2}{\Delta S^2} + r\right),$$
$$C_i = \frac{\Delta t}{2}\left(\frac{\sigma^2 S_i^2}{\Delta S^2} + \frac{(r-q)S_i}{\Delta S}\right).$$

This is implemented in `solvers/Solver.cpp` with the variables `A`, `B`, `C`.

3.3 Boundary conditions

At each time layer t_n, boundaries are imposed through the product interface:

$$V_0^n = \texttt{option.leftBoundary}(t_n), \quad V_{N_s}^n = \texttt{option.rightBoundary}(t_n, S_{\max}).$$

This design makes it straightforward to add products: each product specifies its asymptotic behaviour.

3.4 American early exercise

For American options, after computing the continuation value, we project:

$$V_i^n \leftarrow \max\left(V_i^n,\ \Phi(S_i)\right),$$

implemented via: `if(option.isAmerican()) Vnew[i]=max(Vnew[i], option.earlyExerciseValue(Si));`. This is the standard approach for PDE-based American pricing.

3.5 Outputs of the solver

The solver returns a `Result` struct (see `solvers/ExplicitFdSolver.hpp`) containing:

- `V0`: the full vector $\{V(0, S_i)\}_{i=0}^{N_s}$,
- `price`: interpolated $V(0, S_0)$,
- `delta`, `gamma`: computed at S_0 using finite differences.

4 Implemented products (detailed)

Before discussing grid automation and numerical tuning, we present the set of derivatives supported by our engine. Each product implements the common interface in `products/InterfaceProducts.hpp`:

- `maturity()` and `strike()` (or a representative strike),
- `payoff(S)` (terminal condition),
- `leftBoundary(t)` and `rightBoundary(t, Smax)`,
- optional American hooks: `isAmerican()`, `earlyExerciseValue(S)`.

4.1 European Call (`products/EuropeanCall.hpp`)

Inputs: strike K, maturity T, model (r, σ, q).
Payoff:

$$\Phi(S) = \max(S - K, 0).$$

Boundaries:

$$V(t, 0) = 0, \qquad V(t, S_{\max}) \approx S_{\max} e^{-q(T-t)} - K e^{-r(T-t)}.$$

4.2 European Put (`products/EuropeanPut.hpp`)

Inputs: $K, T, (r, \sigma, q)$.
Payoff:

$$\Phi(S) = \max(K - S, 0).$$

Boundaries:

$$V(t, 0) \approx K e^{-r(T-t)}, \qquad V(t, S_{\max}) \approx 0.$$

4.3 American Put (`products/AmericanPut.hpp`)

Payoff: $\max(K - S, 0)$.
Early exercise: enabled via `isAmerican()=true`, with:

$$V(t, S) \geq \max(K - S, 0).$$

Left boundary choice: we set $V(t, 0) = K$. This reflects the fact that when $S = 0$, immediate exercise yields K and is optimal.

4.4 American Call (`products/AmericanCall.hpp`)

Payoff: $\max(S - K, 0)$.
Early exercise: enabled. In Black–Scholes, early exercise for calls is:

- never optimal when $q = 0$ (American call equals European call),
- potentially optimal when $q > 0$ (dividends can make early exercise attractive).

Boundary at $S_{\max}$: same asymptotic as European call:

$$V(t, S_{\max}) \approx S_{\max} e^{-q(T-t)} - K e^{-r(T-t)}.$$

4.5 Forward/Future-like contract (`products/Future.hpp`)

Inputs: delivery price K, maturity T.
Payoff:

$$\Phi(S) = S - K.$$

This product admits a closed-form price:

$$V(0, S_0) = S_0 e^{-qT} - K e^{-rT}.$$

Boundaries:

$$V(t, 0) \approx -K e^{-r(T-t)}, \qquad V(t, S_{\max}) \approx S_{\max} e^{-q(T-t)} - K e^{-r(T-t)}.$$

4.6 Bull Call Spread (`products/BullCallSpread.hpp`)

Inputs: strikes $K_1 < K_2$, maturity T.
Payoff:

$$\Phi(S) = \max(S - K_1, 0) - \max(S - K_2, 0).$$

This payoff is bounded between 0 and $(K_2 - K_1)$.
Right boundary: for very large S, the payoff saturates to $K_2 - K_1$, hence:

$$V(t, S_{\max}) \approx (K_2 - K_1)e^{-r(T-t)}.$$

Left boundary: $V(t, 0) = 0$.

4.7 Bear Put Spread (`products/BearPutSpread.hpp`)

Inputs: $K_1 < K_2$, T.
Payoff:

$$\Phi(S) = \max(K_2 - S, 0) - \max(K_1 - S, 0),$$

also bounded between 0 and $(K_2 - K_1)$.
Left boundary: for $S = 0$, payoff saturates to $K_2 - K_1$, so:

$$V(t, 0) \approx (K_2 - K_1)e^{-r(T-t)}.$$

Right boundary: $V(t, S_{\max}) \approx 0$.

4.8 Straddle (`products/Straddle.hpp`)

Inputs: strike K, maturity T.
Payoff:

$$\Phi(S) = |S - K| = \max(S - K, 0) + \max(K - S, 0).$$

A key identity is:

$$\text{Straddle} = \text{Call}(K) + \text{Put}(K).$$

Boundaries:

$$V(t, 0) \approx Ke^{-r(T-t)}, \qquad V(t, S_{\max}) \approx S_{\max}e^{-q(T-t)} - Ke^{-r(T-t)}.$$

5 Finite-difference grid and interpolation

5.1 Grid definition (`grid/FdGrid.hpp`)

We discretize:

$$t_n = n\Delta t, \quad n = 0, \ldots, N_t, \qquad S_i = S_{\min} + i\Delta S, \quad i = 0, \ldots, N_s.$$

The `FdGrid` class stores:

$$T,\ S_{\min},\ S_{\max},\ N_t,\ N_s,\ \Delta t,\ \Delta S,$$

and exposes `timeGrid()`, `priceGrid()`, plus a safe linear interpolation routine `interpolate(V, S0)` to evaluate $V(0, S_0)$ even when S_0 is not exactly a grid node.

5.2 Interpolation used for the final price

After solving for the vector $V(0, S_i)$ on the grid, we compute:

$$V(0, S_0) \approx (1-w)V(0, S_i) + wV(0, S_{i+1}), \quad w = \frac{S_0 - S_i}{S_{i+1} - S_i},$$

with (S_i, S_{i+1}) such that $S_i \le S_0 \le S_{i+1}$. This is implemented in `FdGrid::interpolate`.

6 Automatic grid parameter selection: stability vs accuracy vs speed

A key engineering challenge of this project was to select grid parameters that simultaneously ensure:

- **stability** (explicit scheme $\Rightarrow$ CFL-type constraint),
- **accuracy** (small discretization error near S_0 and around the strike),
- **efficiency** (runtime and memory remain reasonable for interactive use).

In practice, this was the main difficulty: a naive grid choice can easily produce a solver that is either (i) unstable and diverges, or (ii) stable but too slow, or (iii) fast but inaccurate because the spatial mesh becomes too coarse.

6.1 Domain truncation: choosing $S_{\max}$ via a lognormal quantile

Under Black–Scholes, $\log S_T$ is Gaussian:

$$\log S_T \sim \mathcal{N}\left(\log S_0 + (r - q - \tfrac{1}{2}\sigma^2)T,\ \sigma^2 T\right).$$

We choose the upper boundary as a high quantile of the lognormal distribution:

$$S_{\max} = S_0 \exp\left((r - q - \tfrac{1}{2}\sigma^2)T + z\,\sigma\sqrt{T}\right),$$

with a fixed safety level z (e.g. $z = 5$). The motivation is to make the probability of $S_T > S_{\max}$ negligible, so that boundary conditions do not contaminate the solution in the region of interest (near S_0). We set $S_{\min} = 0$ for all products.

6.2 Space resolution: choosing N_s from a relative mesh size

A first implementation used fixed values of N_s (e.g. 250/400/800). This worked on some parameter sets but degraded when $S_{\max}$ increased (large T or large σ). Since

$$\Delta S = \frac{S_{\max} - S_{\min}}{N_s},$$

a larger $S_{\max}$ mechanically increases ΔS and reduces accuracy around S_0.

To fix this, our final design makes the spatial step proportional to the spot:

$$\Delta S = \texttt{rel\_dS} \cdot S_0,$$

where `rel_dS` is chosen by the user at runtime (typical values: 0.004 fast, 0.002 balanced, 0.001 accurate). We then set:

$$N_s = \left\lceil \frac{S_{\max} - S_{\min}}{\Delta S} \right\rceil .$$

This ensures that local resolution near S_0 remains comparable across parameter sets, even when $S_{\max}$ varies substantially.

6.3 Time resolution: explicit stability constraint and its implications

The explicit finite-difference scheme is only conditionally stable, so Δt must satisfy a CFL-type bound. A conservative practical constraint is:

$$\Delta t \lesssim \frac{1}{\sigma^2 S_{\max}^2 / \Delta S^2 + r}.$$

We choose Δt using this worst-case bound (with a safety factor) and set:

$$N_t = \left\lceil \frac{T}{\Delta t} \right\rceil .$$

6.4 Main criticism of the approach

The above procedure works reliably for many cases, but it also highlights the main limitation of our numerical method:

- Because we use an **explicit** scheme, stability can force N_t to become very large when σ or $S_{\max}$ is large, which may make the solver slow.
- Using a uniform grid in S can still be inefficient: accuracy is needed mainly near S_0 (and around strikes), while far-away regions consume grid points but contribute little to the price at S_0.

These limitations are well-known in PDE pricing: implicit schemes (Backward Euler / Crank–Nicolson) remove the strict stability constraint, and non-uniform grids (or a change of variable to $\log S$) concentrate points where they matter most. We chose the explicit scheme for simplicity and transparency, but grid automation was crucial to obtain a solver that is both reasonably fast and accurate in practice.

7 Greeks: formulas and numerical computation

7.1 Definitions (continuous-time)

For a derivative price $V(0, S_0; r, \sigma, q, T, \dots)$, the standard Greeks are:

$$\Delta = \frac{\partial V}{\partial S_0}, \quad \Gamma = \frac{\partial^2 V}{\partial S_0^2}, \quad \Theta = \frac{\partial V}{\partial t},$$

$$\nu \text{ (Vega)} = \frac{\partial V}{\partial \sigma}, \quad \rho = \frac{\partial V}{\partial r}.$$

In this project, we compute **Delta** and **Gamma** directly from the PDE solution on the S-grid. The other Greeks (Theta, Vega, Rho) can be computed by bump-and-revalue, but we chose not to implement them to keep the engine focused.

7.2 Discrete formulas used in our code

Let $V_i \approx V(0, S_i)$ and choose $i^\star$ such that $S_{i^\star}$ is the closest grid node to S_0 (while avoiding boundaries). Then:

$$\Delta(0, S_0) \approx \frac{V_{i^\star+1} - V_{i^\star-1}}{2\Delta S},$$

$$\Gamma(0, S_0) \approx \frac{V_{i^\star+1} - 2V_{i^\star} + V_{i^\star-1}}{\Delta S^2}.$$

We avoid $i^\star = 0$ and $i^\star = N_s$ to prevent unreliable boundary derivatives; this is implemented in `solvers/Solver.cpp` by shifting the index away from the boundaries.

8 Test suite: consistency checks

The test executable `bs_tests` runs a suite of checks comparing our numerical outputs to basic financial identities and numerical properties:

- **Put–call parity (European):**

$$C - P = S_0 e^{-qT} - Ke^{-rT}.$$

- **Forward closed-form price:**

$$V_{\text{fwd}}(0) = S_0 e^{-qT} - Ke^{-rT}.$$

- **American dominance:** $P^{Am} \geq P^{Eu}$ and $C^{Am} \geq C^{Eu}$.
- **Straddle consistency:** Straddle $\approx C + P$.
- **Spread bounds:** spreads are bounded by $(K_2 - K_1)e^{-rT}$.
- **Greek sanity checks:** for forwards, $\Delta \approx e^{-qT}$ and $\Gamma \approx 0$.

These tests were crucial to ensure our code behaves correctly and to detect regressions after refactoring.

9 How to build and run the code

The full source code is publicly available on GitHub:

`https://github.com/Samgit0532/Pricing_par_EDP`.

This section briefly explains how to compile and run the project, either using `CMake` or directly with `g++`. A README.md is also there for more details.

9.1 Build and run with CMake

9.1.1 Requirements

- `CMake` version ≥ 3.16,
- a C++ compiler supporting the `C++17` standard (e.g. `g++`).

9.1.2 Build

From the root directory of the project, run:

```
cmake -S . -B build
cmake --build build -j
```

This generates two executables in the `build/` directory.

9.1.3 Run the interactive application

```
./build/bs_app
```

The program launches an interactive interface where the user selects:

- the financial product,
- market parameters (S_0, r, σ, q),
- contract parameters (maturity, strike(s)),
- the numerical resolution parameter `rel_dS`.

9.1.4 Run the test suite

```
./build/bs_tests
```

The test executable automatically checks pricing identities and numerical sanity properties (put–call parity, forward pricing, American dominance, bounded spreads, and Greek consistency).

9.2 Build and run without CMake (direct compilation)

9.2.1 Requirements

- `g++` with `C++17` support.

9.2.2 Compile the interactive application

```
g++ -std=c++17 -O2 -O2 -I./src src/main.cpp src/solvers/Solver.cpp -o bs_app
```

Run:

```
./bs_app
```

9.2.3 Compile the test executable

```
g++ -std=c++17 -O2 -I./src src/tests/TestPricing.cpp src/solvers/Solver.cpp -o bs_tests
```

Run:

```
./bs_tests
```