RAPPORT DE PROJET

Informatique pour la Finance Quantitative

Pricing d'Options Américaines par Monte Carlo Longstaff-Schwartz

Auteurs :

Issam Fradi & Yassine Mannai

Tuteur :

Emmanuel Fruchard

Université Paris Dauphine – PSL

Mars 2026

Contents

1 Introduction

Ce rapport présente l'implémentation complète d'un pricer d'options américaines et européennes utilisant trois méthodes de pricing complémentaires :

- **Monte Carlo Longstaff-Schwartz (LSM)** : simulation stochastique avec régression polynomiale pour l'exercice anticipé
- **Arbre Trinomial** : modèle discret avec backward induction
- **Black-Scholes** : solution analytique pour les options européennes

Le projet intègre une interface interactive Streamlit permettant de comparer les trois méthodes en temps réel, avec calcul des Greeks, analyse de convergence, et visualisation de la régression Longstaff-Schwartz.

1.1 Paramètres de référence

Sauf mention contraire, les résultats de ce rapport utilisent les paramètres suivants :

Paramètre	**Symbole**	**Valeur**
Spot initial	S_0	100
Strike	K	100
Taux sans risque	r	5%
Volatilité	σ	20%
Maturité	T	1 an
Type d'option	–	Put Américain

Table 1 – Paramètres de référence

2 Architecture du Code

2.1 Structure du projet

Le projet suit une architecture modulaire organisée en package Python :

```
MC-Longstaff-Schwartz/
|-- pricing/                        # Package principal
|   |-- __init__.py                 # Re-exports compatibilite
|   |-- parameters/
|   |   |-- market.py               # Parametres marche (validation)
|   |   +-- option.py               # Definition option (validation)
|   |-- brownian_motion/
|   |   +-- brownian_motion.py      # Generation chemins browniens
|   |-- models/
|   |   |-- model.py                # Classe abstraite Model
|   |   |-- black_scholes/
|   |   |   +-- blackscholes.py     # Pricer Black-Scholes
|   |   |-- lsm/
|   |   |   |-- model_mc.py         # Facade MonteCarloPricer
|   |   |   |-- pricing_mixins.py   # Mixins (EU, US, batch, ...)
|   |   |   |-- regression.py       # Bases polynomiales + LSRegressor
|   |   |   |-- path_generator.py   # Generation chemins GBM
|   |   |   +-- static_pricing.py   # Fonctions statiques
|   |   +-- tree/
|   |       |-- trinomial_tree.py   # Arbre trinomial
```

```
|   |       +-- node.py              # Noeud de l'arbre
|   |-- greeks/
|   |   +-- greeks.py                # GreeksCalculator (bump-and-reprice)
|   +-- others/
|       +-- funcs.py                 # Utilitaires arbre
|-- tests/                           # Tests unitaires (pytest, 126 tests)
|-- rapport/                         # Rapport LaTeX
|-- streamlit_app.py                 # Interface interactive
+-- pyproject.toml                   # Configuration
```

2.2 Hiérarchie des classes

L'architecture repose sur un pattern **façade + mixins** avec une classe abstraite `Model` :

```python
class Model(ABC):
    """Classe abstraite pour tous les pricers."""
    def __init__(self, pricing_date: dt.date):
        self.pricing_date = pricing_date

class MonteCarloPricer(
    EuropeanPricerMixin,    # Pricing europeen
    AmericanLSMixin,        # Longstaff-Schwartz
    ScalarPricerMixin,      # Mode scalaire
    BatchPricerMixin,       # Traitement par batch
    ExerciseFrontierMixin,  # Frontiere d'exercice
    ConvergenceMixin,       # Analyse convergence
    Model,
):
    """Facade : delegue a 6 mixins specialises."""

class TrinomialTree(Model):
    """Pricing par arbre trinomial recombinant."""
```

Le pricer Monte Carlo est décomposé en plusieurs modules spécialisés :

- `model_mc.py` : façade légère (∼150 lignes) — routeur, constructeur
- `pricing_mixins.py` : 6 mixins (européen, américain LS, scalaire, batch, frontière, convergence)
- `regression.py` : 5 bases polynomiales + régresseur par moindres carrés
- `path_generator.py` : génération de chemins GBM avec dividende discret
- `static_pricing.py` : fonctions de pricing sans instanciation

Les Greeks sont calculés par un composant dédié `GreeksCalculator` (bump-and-reprice), indépendant du pricer utilisé.

2.3 Objets de données

Les paramètres de marché et d'option sont encapsulés dans des `dataclass` avec validation :

```python
@dataclass
class Market:
    stock_price: float    # Spot S0
```

```
    int_rate: float       # Taux sans risque r
    sigma: float          # Volatilite annuelle
    div: float = 0.0      # Dividende discret
    div_date: Optional[dt.date] = None

    def __post_init__(self):
        if self.stock_price <= 0:
            raise ValueError("stock_price must be > 0")
        if self.sigma < 0:
            raise ValueError("sigma must be >= 0")

@dataclass
class Option:
    K: float              # Strike
    maturity: dt.date     # Date de maturite
    option_type: str      # 'call' ou 'put'
    option_class: str     # 'european' ou 'american'

    def __post_init__(self):
        if self.K <= 0:
            raise ValueError("K must be > 0")
        if self.option_type not in ("call", "put"):
            raise ValueError("option_type: 'call'/'put'")
```

3 Simulation Monte Carlo – Mouvement Brownien Géométrique

3.1 Modèle théorique

Le prix du sous-jacent suit un mouvement brownien géométrique (GBM) sous la mesure risque-neutre :

$$dS_t = r\, S_t\, dt + \sigma\, S_t\, dW_t \tag{1}$$

dont la solution exacte est :

$$S_{t+\Delta t} = S_t \cdot \exp\left[\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\, Z\right], \quad Z \sim \mathcal{N}(0,1) \tag{2}$$

3.2 Génération des chemins

L'implémentation utilise le module `PathGenerator` pour générer les chemins de manière vectorisée :

```
class PathGenerator:
    """Generation de chemins GBM vectorises."""

    def _build_gbm_paths(self, dW, dt_val):
        drift = (self.market.int_rate
                 - 0.5 * self.market.sigma**2) * dt_val
        diffusion = self.market.sigma * dW
        log_returns = np.cumsum(drift + diffusion, axis=1)

        S_paths = np.zeros((dW.shape[0],
                            self.nb_steps + 1))
        S_paths[:, 0] = self.market.stock_price
        S_paths[:, 1:] = (self.market.stock_price
                          * np.exp(log_returns))
        return S_paths, log_returns
```

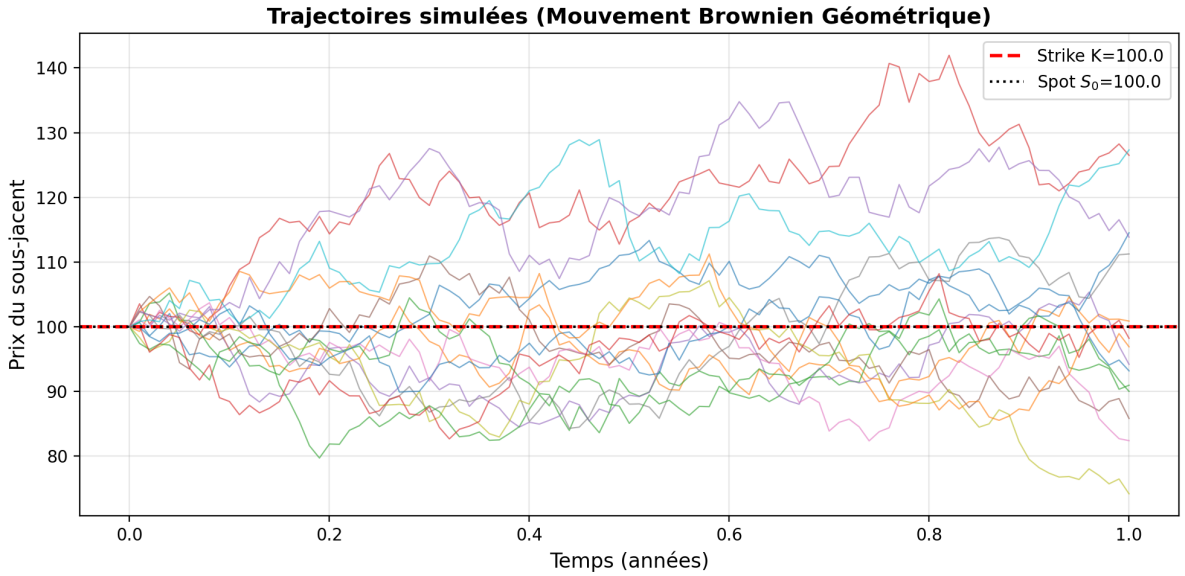

Figure 1 – 15 trajectoires simulées du sous-jacent par GBM ($S_0 = 100$, $\sigma = 20\%$, $T = 1$ an)

3.3 Réduction de variance : variables antithétiques

Pour chaque vecteur gaussien Z, on génère également le chemin avec $-Z$:

$$\hat{P} = \frac{1}{2}\left[f(Z) + f(-Z)\right] \tag{3}$$

```
def generate_dw(self, T, nb_steps, nb_paths, antithetic=True):
    dt_val = T / nb_steps
    n = nb_paths // 2 if antithetic else nb_paths
    Z = self.rng.standard_normal((n, nb_steps))
    if antithetic:
        Z = np.concatenate([Z, -Z], axis=0)
    return Z * np.sqrt(dt_val), dt_val
```

3.4 Traitement du dividende discret

Le dividende discret est appliqué au pas de temps correspondant à sa date de détachement :

$$S^{\text{ex}}_{t_{\text{div}}} = S^{\text{cum}}_{t_{\text{div}}} - D \tag{4}$$

Les chemins post-dividende sont reconstruits à partir du prix ex-dividende pour maintenir la cohérence des log-returns.

4 Algorithme de Longstaff-Schwartz

4.1 Principe

L'algorithme de Longstaff & Schwartz (2001) résout le problème d'exercice anticipé des options américaines par une **récursion backward** combinée à une **régression polynomiale** pour estimer la valeur de continuation.

À chaque pas de temps t, pour les chemins *in-the-money* (ITM), on estime :

$$\hat{C}(S_t) = \mathbb{E}\left[e^{-r\Delta t} \cdot V_{t+1} \,\middle|\, S_t\right] \approx \sum_{k=0}^{d} \beta_k \, \phi_k(S_t) \tag{5}$$

où $\{\phi_k\}$ est une base de fonctions polynomiales et les β_k sont estimés par moindres carrés.

La décision d'exercice est alors :

$$\text{Exercer en } t \quad \Longleftrightarrow \quad h(S_t) > \hat{C}(S_t) \tag{6}$$

où $h(S_t) = \max(K - S_t, 0)$ pour un put.

4.2 Implémentation de la récursion backward

```
def _ls_backward_recursion(self, S, cashflows,
                           discount_factor,
                           regression_type, degree):
    """Backward recursion (from T-1 to 1)."""
    for t in range(self.nb_steps - 1, 0, -1):
        cashflows = cashflows * discount_factor
        S_t = S[:, t]
        intrinsic_val = self.option.payoff(S_t)
        # Only consider In-The-Money paths
        itm = intrinsic_val > 0
        if np.count_nonzero(itm) > 0:
            cashflows = self._ls_update_cashflows(
                cashflows, S_t, intrinsic_val,
                itm, regression_type, degree
            )
    return cashflows
```

```
def _ls_update_cashflows(self, cashflows, S_t,
                         intrinsic_val, itm,
                         regression_type, degree):
    """Decision d'exercice a chaque pas de temps."""
    X, Y = S_t[itm], cashflows[itm]
    continuation_val = self._compute_continuation_value(
        X, Y, regression_type, degree
    )
    exercise = intrinsic_val[itm] > continuation_val
    updated_cf = cashflows.copy()
    updated_cf[itm] = np.where(
        exercise, intrinsic_val[itm], Y
    )
    return updated_cf
```

4.3 Régression et valeur de continuation

La valeur de continuation est estimée par régression par moindres carrés via la classe `LSRegressor` :

```
class LSRegressor:
    def compute_continuation_value(self, X, Y,
            regression_type, degree):
        if len(X) < degree + 1:
            return Y
        basis = BasisBuilder.build(X, regression_type,
                                   degree)
        return self._fit_regression(basis, X, Y, degree)

    def _fit_regression(self, basis, X, Y, degree):
```

```
        try:
            coeffs, _, _, _ = np.linalg.lstsq(
                basis, Y, rcond=None)
            return basis @ coeffs
        except np.linalg.LinAlgError:
            coeffs = np.polyfit(X, Y, min(degree, 2))
            return np.polyval(coeffs, X)
```

4.4 Visualisation de la régression

La figure 4.4 montre la régression Longstaff-Schwartz à trois pas de temps différents. On observe que :

- Les **points bleus** représentent les cashflows actualisés (variable dépendante Y)
- La **courbe rouge** est la régression polynomiale $\hat{C}(S_t)$
- La **droite verte** est la valeur intrinsèque $h(S_t) = \max(K - S_t, 0)$
- L'exercice est optimal là où la courbe verte dépasse la courbe rouge

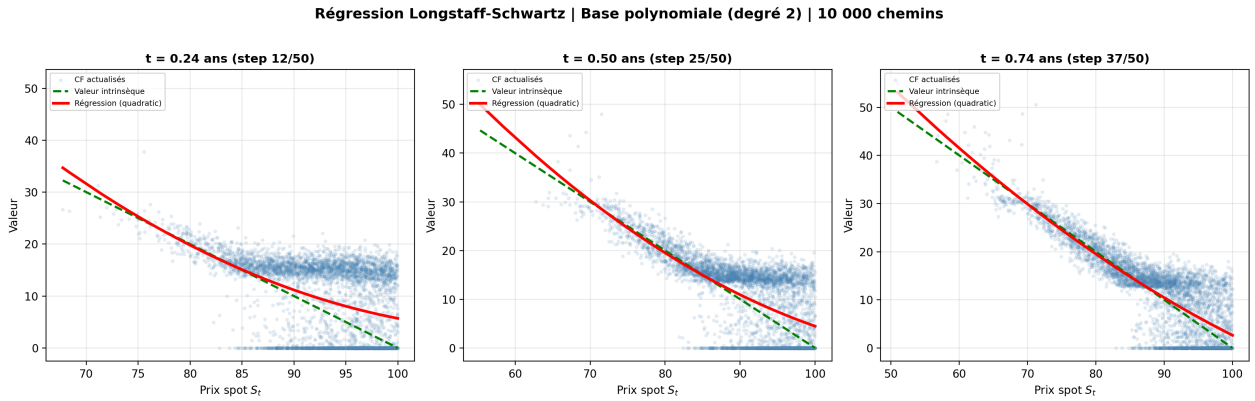

Figure 2 – Régression Longstaff-Schwartz à trois pas de temps (base polynomiale, degré 2, 10 000 chemins). La courbe rouge (régression) estime la valeur de continuation $\hat{C}(S_t)$.

4.5 Bases de régression

Cinq bases polynomiales sont implémentées pour la régression :

Base	Fonctions	Récurrence
Quadratique	$1, x, x^2, \ldots, x^d$	$\phi_k(x) = x^k$
Laguerre	$L_0 = 1,\ L_1 = 1 - x$	$L_n = \frac{(2n-1-x)L_{n-1} - (n-1)L_{n-2}}{n}$
Hermite	$H_0 = 1,\ H_1 = x$	$H_n = xH_{n-1} - (n-1)H_{n-2}$
Legendre	$P_0 = 1,\ P_1 = \tilde{x}$	$P_n = \frac{(2n-1)\tilde{x}P_{n-1} - (n-1)P_{n-2}}{n}$
Chebyshev	$T_0 = 1,\ T_1 = \tilde{x}$	$T_n = 2\tilde{x}T_{n-1} - T_{n-2}$

Table 2 – Bases de régression implémentées ($\tilde{x} = 2\frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1$)

```
class BasisBuilder:
    """Construction des bases polynomiales."""

```

```
    @staticmethod
    def build(X, regression_type, degree):
        """Dispatch vers la bonne base."""
        methods = {
            "quadratic": BasisBuilder._quadratic,
            "laguerre":  BasisBuilder._laguerre,
            "hermite":   BasisBuilder._hermite,
            "legendre":  BasisBuilder._legendre,
            "chebyshev": BasisBuilder._chebyshev,
        }
        return methods[regression_type](X, degree)

    @staticmethod
    def _laguerre(X, degree):
        n = len(X)
        basis = np.zeros((n, degree + 1))
        basis[:, 0] = 1.0
        if degree >= 1:
            basis[:, 1] = -X + 1
        if degree >= 2:
            basis[:, 2] = 0.5*(X**2 - 4*X + 2)
        for i in range(3, degree + 1):
            basis[:, i] = (
                (2*(i-1)+1-X)*basis[:,i-1]
                - (i-1)*basis[:,i-2]
            ) / i
        return basis
```

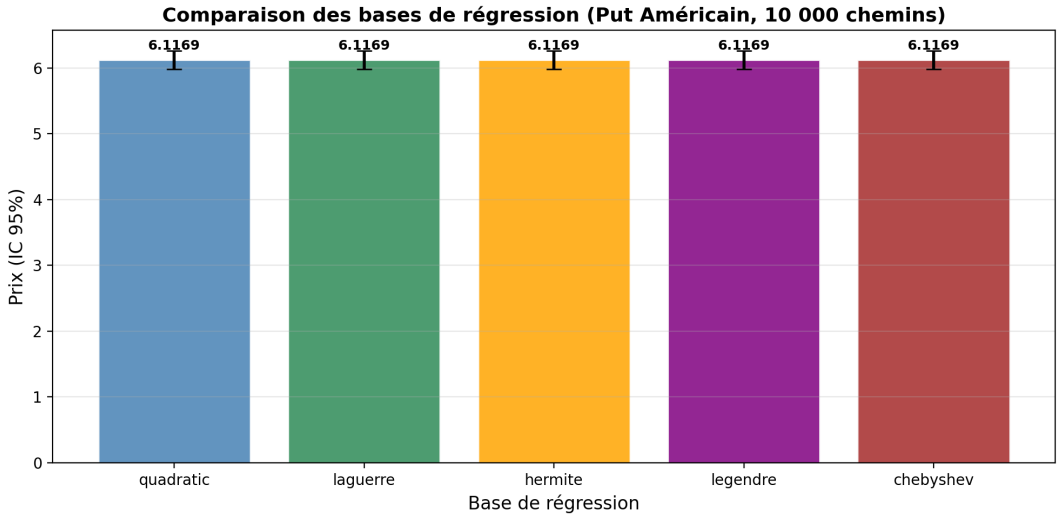

Figure 3 – Comparaison des prix obtenus avec les 5 bases de régression (Put Américain, 10 000 chemins). Les barres d'erreur représentent l'IC à 95%.

Observation : Les cinq bases produisent des prix très proches, confirmant la robustesse de l'algorithme LSM vis-à-vis du choix de base. La base quadratique standard reste le choix par défaut en raison de sa simplicité.

5 Frontière d'Exercice Optimale

L'algorithme LSM permet d'extraire la **frontière d'exercice optimale** : le prix critique $S^*(t)$ en dessous duquel (pour un put) il est optimal d'exercer l'option.

```python
def get_exercise_frontier(self, antithetic=True,
        regression_type="quadratic", degree=2):
    S, dt_val, T = self._generate_price_paths(antithetic)
    discount_factor = np.exp(-self.market.int_rate * dt_val)
    times = np.linspace(0, T, self.nb_steps + 1)
    frontier_prices = np.full(self.nb_steps + 1, np.nan)
    frontier_prices[-1] = self.option.K
    # Backward recursion
    cashflows = self.option.payoff(S[:, -1])
    cashflows, frontier_prices = self._frontier_backward(
        S, cashflows, discount_factor, frontier_prices,
        regression_type, degree
    )
    return times, frontier_prices
```

La frontière est déterminée à chaque pas en identifiant le prix maximal parmi les chemins où l'exercice est déclenché :

```python
def _update_frontier(self, frontier_prices, t,
                     X, exercise):
    if np.any(exercise) and np.any(~exercise):
        exercising_prices = X[exercise]
        if self.option.option_type == "put":
            frontier_prices[t] = np.max(exercising_prices)
        else:
            frontier_prices[t] = np.min(exercising_prices)
```

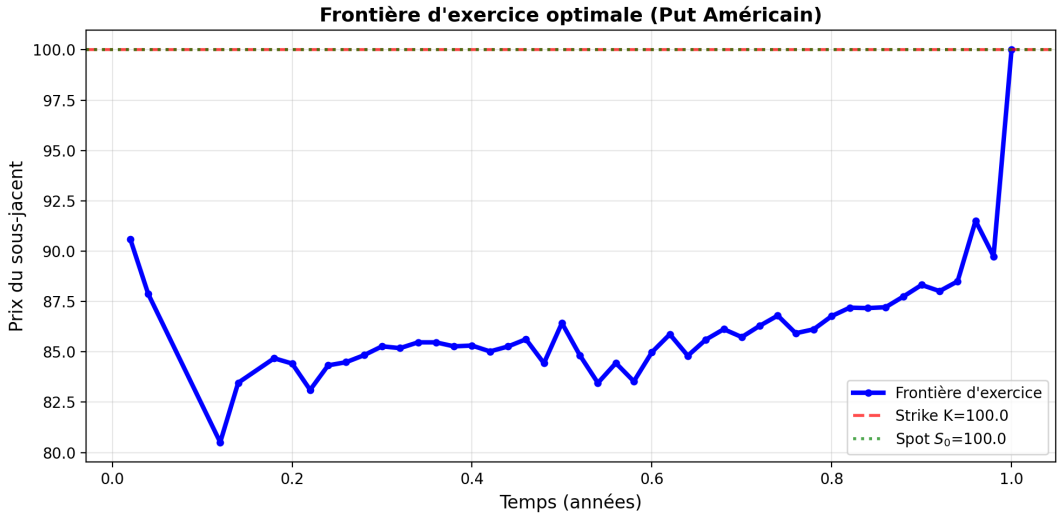

Figure 4 – Frontière d'exercice optimale pour un put américain. L'exercice est optimal lorsque S_t passe en dessous de la frontière.

Interprétation : Pour un put américain ATM, la frontière d'exercice se situe en dessous du strike. Elle est croissante avec le temps restant à maturité : plus on est proche de l'échéance, plus le seuil d'exercice remonte vers K.

6 Analyse de Convergence

6.1 Convergence du prix MC

La convergence Monte Carlo est en $O(1/\sqrt{N})$:

$$\text{Erreur standard} = \frac{\hat{\sigma}}{\sqrt{N}} \tag{7}$$

où $\hat{\sigma}$ est l'écart-type des payoffs actualisés.

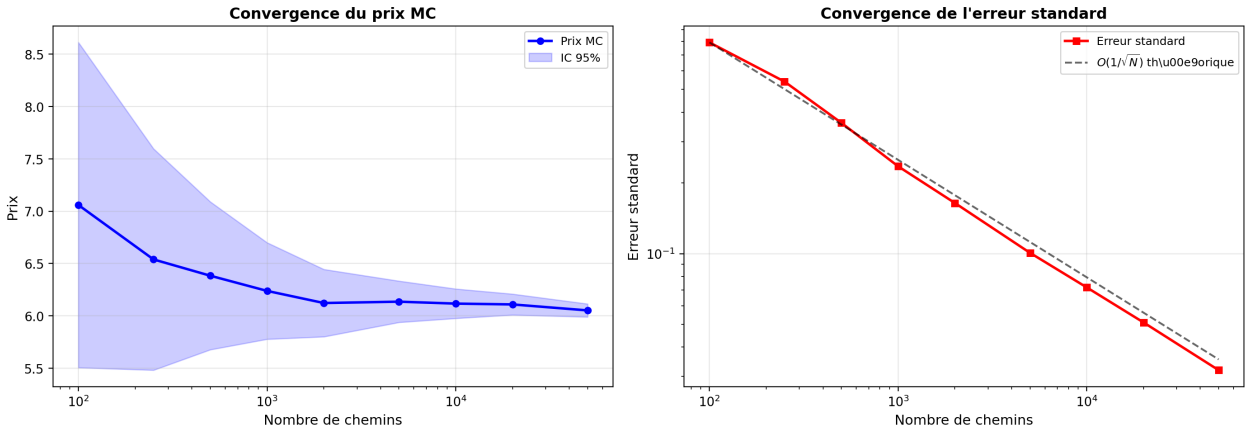

Figure 5 – Convergence du prix MC (gauche) et de l'erreur standard (droite). La ligne pointillée noire montre la décroissance théorique en $O(1/\sqrt{N})$.

6.2 Distribution des payoffs

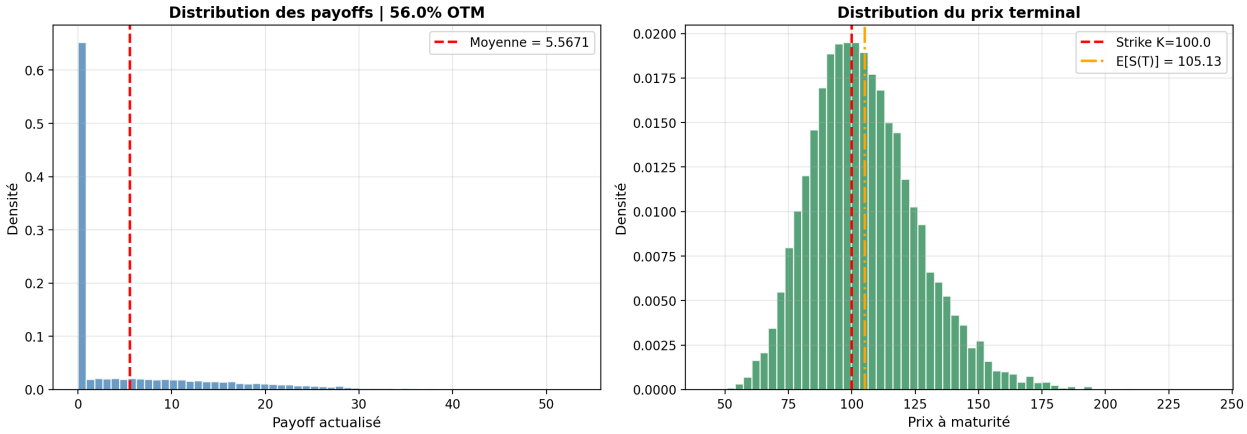

Figure 6 – Distribution des payoffs actualisés (gauche) et du prix terminal (droite). Le pic en zéro correspond aux chemins OTM à maturité.

Observations :

- La distribution des payoffs est **fortement asymétrique** (skewed) avec une masse importante en zéro (chemins OTM).
- La distribution terminale de S_T suit une **loi log-normale** centrée autour de $S_0 e^{rT} \approx 105.13$, conformément au GBM.

7 Comparaison des Méthodes de Pricing

7.1 Prix en fonction du strike

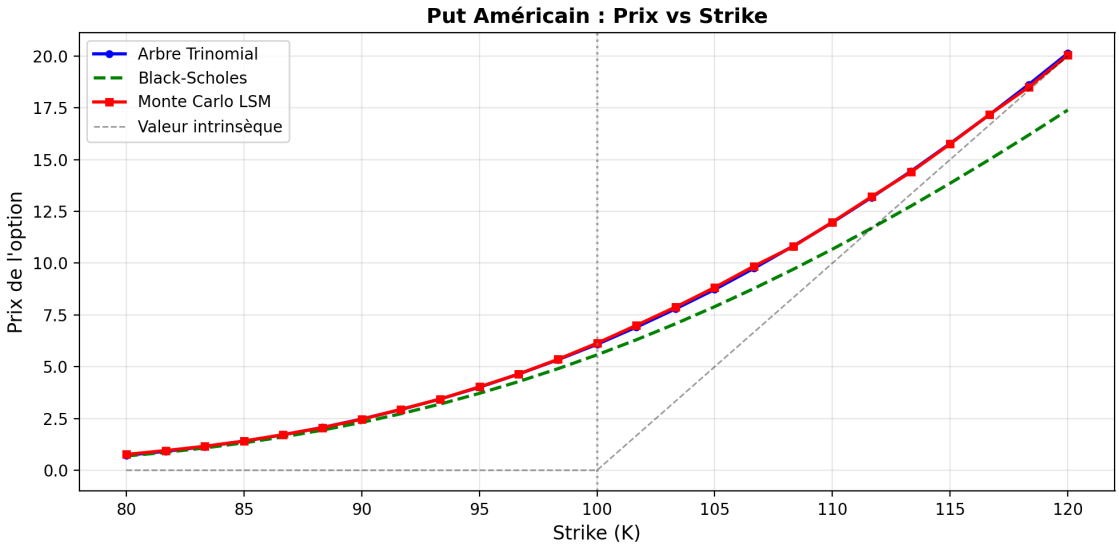

Figure 7 – Comparaison des prix (Put Américain) en fonction du strike : arbre trinomial, Black-Scholes (européen), et Monte Carlo LSM.

Observations :

- Le put américain (Tree, MC) est toujours **supérieur ou égal** au put européen (BS), conformément à la théorie : $V_{\text{am}} \geq V_{\text{eu}}$.
- La prime d'exercice anticipé est plus visible pour les options deep ITM (strike élevé).
- Tree et MC convergent vers des prix très proches, validant mutuellement les implémentations.

7.2 Tableau comparatif des méthodes

Critère	Black-Scholes	Arbre Trinomial	MC LSM
Options européennes	**OK**	**OK**	**OK**
Options américaines	**X**	**OK**	**OK**
Dividende discret	Approx.	Exact	Exact
Solution analytique	**OK**	**X**	**X**
Convergence	Exacte	$O(1/N)$	$O(1/\sqrt{N})$
Complexité	$O(1)$	$O(N^2)$	$O(M \cdot N)$
Greeks	Analytiques	Arbre	Diff. finies
Visualisation régression	–	–	**OK**

Table 3 – Comparaison des trois méthodes de pricing

8 Calcul des Greeks

8.1 Méthodes de calcul

Les Greeks sont calculés de manière unifiée par la classe `GreeksCalculator` (bump-and-reprice), qui fonctionne avec n'importe quel pricer :

Greek	Méthode	Formule
Δ	Différences finies centrales	$\frac{f(S+h)-f(S-h)}{2h}$
Γ	Dérivée seconde	$\frac{f(S+h)+f(S-h)-2f}{h^2}$
$\mathcal{V}$	Bump σ	$\frac{f(\sigma+h)-f(\sigma-h)}{2h}$
Θ	Bump t	$f(t+1j)-f(t)$
ρ	Bump r	$\frac{f(r+h)-f(r-h)}{2h}$
Vanna	Bump croisé S, σ	$\frac{\Delta(\sigma+h)-\Delta(\sigma-h)}{2h}$

Table 4 – GreeksCalculator : méthode unique pour tous les pricers

```python
class GreeksCalculator:
    """Greeks par bump-and-reprice, generique."""

    def __init__(self, price_fn, market, pricer=None):
        self.price_fn = price_fn
        self.market = market
        self.pricer = pricer  # optionnel, pour theta

    def delta(self, bump=0.01):
        h = self.market.stock_price * bump
        self.market.stock_price += h
        p_up = self.price_fn()
        self.market.stock_price -= 2 * h
        p_down = self.price_fn()
        self.market.stock_price += h  # restore
        return (p_up - p_down) / (2 * h)

    def greeks(self):
        """Renvoie un dict complet de tous les Greeks."""
        return {"delta": self.delta(), "gamma": ...,
                "vega": ..., "theta": ..., "rho": ...}
```

8.2 Sensibilité des Greeks au prix spot

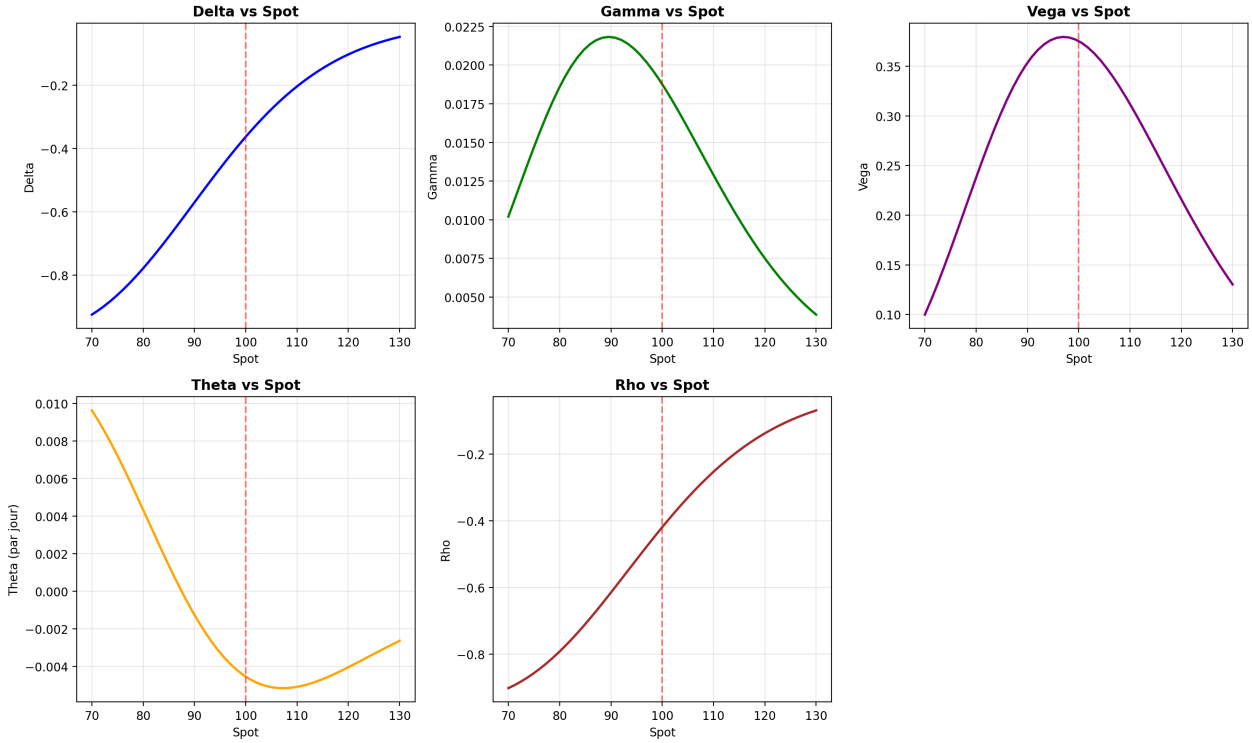

Figure 8 – Sensibilité des Greeks au prix spot (Put Européen, calcul analytique BS). La ligne rouge verticale indique le spot courant $S_0 = 100$.

Interprétation :

- **Delta** : négatif pour un put, tend vers -1 en deep ITM et 0 en OTM
- **Gamma** : maximum ATM ($S \approx K$), symétrique, mesure la convexité
- **Vega** : maximum ATM, l'option est plus sensible à σ quand $S \approx K$
- **Theta** : toujours négatif (décroissance temporelle), minimum ATM
- **Rho** : négatif pour un put (hausse des taux $\Rightarrow$ baisse du prix)

8.3 Optimisation du calcul

Le `GreeksCalculator` fonctionne avec n'importe quelle callable `price_fn`. Pour les options européennes, on peut passer le pricer BS analytique (∼14 ms) plutôt que MC (∼14 s), ce qui reste transparent pour l'appelant.

9 Interface Interactive Streamlit

9.1 Architecture de l'interface

L'application Streamlit (∼500 lignes) est organisée en 5 onglets :

#	Onglet	Contenu
1	Pricing	Prix Tree/BS/MC, métriques, frontière d'exercice
2	Greeks	Tableau comparatif, courbes de sensibilité
3	Plots	Régression LS, distributions, convergence, benchmarks
4	Tree	Visualisation interactive de l'arbre trinomial
5	Multi-Seed	Analyse statistique sur plusieurs seeds

Table 5 – Onglets de l'interface Streamlit

9.2 Gestion du cache

Streamlit fournit des décorateurs de cache pour éviter les recalculs :

```
@st.cache_data(show_spinner=False)
def bs_price_cached(S, K, T, r, sigma, opt_type, **kw):
    """Cache les prix BS pour eviter les recalculs."""
    bs = BlackScholesPricer(S, K, T, r, sigma,
                            opt_type, **kw)
    return float(bs.price())

@st.cache_resource(show_spinner=False)
def tree_cached(market, N, pruning, epsilon,
                pricing_date):
    """Cache l'arbre trinomial (objet lourd)."""
    return TrinomialTree(market, N=N, pruning=pruning,
                         epsilon=epsilon,
                         pricing_date=pricing_date)
```

Un hash MD5 des paramètres MC détecte automatiquement les changements et force le recalcul :

```
import hashlib
params_str = f"{nb_paths}|{nb_steps}|{regression}|..."
current_hash = hashlib.md5(params_str.encode()).hexdigest()
if current_hash != st.session_state.last_mc_params_hash:
    st.session_state.calculate = False  # Force recalcul
```

10 Performance et Traitement par Batch

10.1 Gestion mémoire

Pour un grand nombre de chemins ($M > 50\,000$), la matrice $S \in \mathbb{R}^{M\times(N+1)}$ peut dépasser la mémoire. Le mode **batch** découpe le calcul en blocs :

```
class BatchPricerMixin:
    """Batch pricing with ThreadPoolExecutor."""

    def _price_batched(self, antithetic, regression_type,
                       degree, american_method):
        bs = self.batch_size
        nb_batches = math.ceil(self.nb_paths / bs)
```

```
        batch_prices = self._run_batches(
            nb_batches, bs, self.nb_paths, antithetic,
            regression_type, degree, american_method)
        return self._aggregate_batch_results(batch_prices)

    def _run_batches(self, nb_batches, bs, total, ...):
        from concurrent.futures import ThreadPoolExecutor
        n_workers = min(nb_batches, os.cpu_count() or 1)
        if n_workers > 1:
            with ThreadPoolExecutor(n_workers) as pool:
                futures = [pool.submit(
                    self._price_single_batch, k, n_k, ...)
                    for k, n_k in batch_specs]
                return [f.result() for f in futures]
        return [self._price_single_batch(k, n_k, ...)
                for k, n_k in batch_specs]
```

10.2 Benchmarks

Méthode	**Configuration**	**Temps**
Black-Scholes	Analytique	< 1 ms
Arbre Trinomial	$N = 100$	~ 30 ms
Arbre Trinomial	$N = 500$	~ 700 ms
Monte Carlo LSM	5 000 chemins	~ 100 ms
Monte Carlo LSM	50 000 chemins	~ 1 s
Greeks BS (30 pts)	Analytique	~ 14 ms
Greeks Tree (30 pts)	Avec bumps	~ 6 s

Table 6 – Temps d'exécution mesurés

11 Tests Unitaires

Le projet utilise `pytest` avec paramétrisation pour valider les résultats :

```
def test_european_call_converges_to_bs():
    """Le prix MC europeen converge vers BS."""
    mc_price = pricer.price(antithetic=True,
                            method="vectoriel")
    bs_price = BlackScholesPricer(S0, K, T, r, sigma,
                                  "call").price()
    assert abs(mc_price - bs_price) / bs_price < 0.01

def test_american_put_geq_european_put():
    """Le put americain >= put europeen."""
    am_price = pricer_am.price(...)
    eu_price = pricer_eu.price(...)
    assert am_price >= eu_price - 1e-6
```

Test	Statut	Tolérance
MC Européen → BS	**OK**	$< 1\%$ relatif
Scalaire = Vectoriel (même seed)	**OK**	$< 10^{-10}$
Antithétique réduit la variance	**OK**	$\sigma_{\text{anti}} < \sigma_{\text{std}}$
Américain ≥ Européen	**OK**	≥ 0
5 bases de régression	**OK**	Convergence

Table 7 – Couverture des tests

12 Synthèse

12.1 Récapitulatif des fonctionnalités

Module	Fonctionnalités clés
`models/lsm/`	Pricing MC vectorisé (façade + 6 mixins), 5 bases de régression, batch mode, variables antithétiques, frontière d'exercice, convergence
`models/tree/`	Arbre trinomial recombinant, pruning, dividende discret, 5 Greeks, visualisation
`models/black_scholes/`	Pricing analytique Black-Scholes
`greeks/`	`GreeksCalculator` générique (bump-and-reprice), 6 Greeks
`parameters/`	`Market` et `Option` avec validation
`streamlit_app.py`	Interface 5 onglets, cache intelligent, 8+ types de graphiques
`tests/`	126 tests paramétrisés avec pytest, validation croisée

Table 8 – Synthèse des fonctionnalités du projet

12.2 Enseignements clés

1. **Architecture modulaire :** la décomposition en façade + mixins + composants spécialisés (`PathGenerator`, `LSRegressor`, `BasisBuilder`, `GreeksCalculator`) améliore la maintenabilité et la testabilité.

2. **Greeks génériques :** le `GreeksCalculator` par bump-and-reprice fonctionne avec tous les pricers sans code spécifique.

3. **Vectorisation NumPy :** l'approche vectorisée (sans boucles Python sur les chemins) est essentielle pour la performance MC.

4. **Choix de la base de régression :** les 5 bases (quadratique, Laguerre, Hermite, Legendre, Chebyshev) produisent des résultats très similaires pour le degré 2.

5. **Variables antithétiques :** réduction significative de la variance (∼30–40%) pour un coût de calcul négligeable.

6. **Validation croisée :** la comparaison Tree vs MC vs BS permet de détecter les bugs d'implémentation de manière fiable.

7. **Optimisation des Greeks :** utiliser BS analytique pour les européennes et interpolation pour les américaines réduit drastiquement le temps de calcul.