Sorbonne Université

Calcul haute performance : programmation et algorithmique avancées

Projects: Merge Sort, Monte Carlo and PDE simulations, and machine learning

The presentation of the results on December 18th, 2025

/!\Submission and Presentation Guidelines common to all projects

Students must submit both their presentation slides and the complete source code of their project to their instructor (by email to L. ABBAS TURKI) no later than December 17th. The submitted code must be readable and well-commented to ensure clarity and facilitate evaluation. The project grade will be based on both the quality of the source code, the oral presentation and the answers to questions, which will take place during the session on December 18th. The total duration of the presentation, including questions, should not exceed 12 minutes and must follow the structure below:

- **5 to 6 minutes: Presentation of results.**
- **1 to 2 minutes: Explanation of the code.**
- **4 to 6 minutes: Questions and discussion.**

The number of slides dedicated to presenting results should be less than six (6) but more than three (3), and the total duration of the presentation excluding questions should be between 6 and 8 minutes. Within each group, speaking time must be evenly distributed among all members. If this balance is not respected, all questions will be addressed exclusively to the student who has spoken the least during the presentation.

Contents

1 Monte Carlo simulation of Heston model

The Heston model for asset pricing has been widely examined in the literature. Under this model, the dynamics of the asset price S_t and the variance v_t are governed by the following system of stochastic differential equations:

$$dS_t = rS_t dt + \sqrt{v_t} S_t d\hat{Z}_t \tag{1}$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t \tag{2}$$
$$\hat{Z}_t = \rho W_t + \sqrt{1-\rho^2} Z_t \tag{3}$$

where:

- the spot values $S_0 = 1$ and $v_0 = 0.1$,
- r is the risk-free interest rate, we assume $r = 0$,
- κ is the mean reversion rate of the volatility,
- θ is the long-term volatility,
- σ is the volatility of volatility,
- W_t and Z_t are independent Brownian motions.

In this project, we aim to compare three distinct methods for simulating an at-the-money call option (where "at-the-money" here means $K = S_0 = 1$) at maturity $T = 1$ under the Heston model. The option has a payoff given by $f(x) = (x-K)_+$, and, thus, we want to simulate with Monte Carlo the expectation $E[f(S_T)] = E[(S_1 - 1)_+]$. This comparison will focus on the efficiency and accuracy of each simulation method in pricing the call option within the stochastic volatility framework of the Heston model.

We begin with the Euler discretization scheme, which updates the asset price S_t and the volatility v_t at each time step as follows:

$$S_{t+\Delta t} = S_t + rS_t\Delta t + \sqrt{v_t} S_t \sqrt{\Delta t}(\rho G_1 + \sqrt{1-\rho^2} G_2) \tag{4}$$
$$v_{t+\Delta t} = g\left(v_t + \kappa(\theta - v_t)\Delta t + \sigma\sqrt{v_t}\sqrt{\Delta t} G_1\right) \tag{5}$$

where G_1 and G_2 are independent standard normal random variables, and the function g is either taken to be equal to $(\cdot)_+$ or to $|\cdot|$.

1. Assuming $\kappa = 0.5$, $\theta = 0.1$, $\sigma = 0.3$ and using a discretization $\Delta t = 1/1000$, write down a Monte Carlo simulation code using Euler discretization to approximate $E[(S_1 - 1)_+]$. (8 points)

As an alternative to the Euler scheme, we are going to implement a version of the exact simulation procedure presented in the paper [4]. The steps that we choose differ a little bit as we take the discretization $\Delta t = 1/1000$ and we do the following:

step 1. Within a for loop on time steps, we define

$$d = 2\kappa\theta/\sigma^2, \quad \lambda = \frac{2\kappa e^{-\kappa\Delta t} v_t}{\sigma^2(1 - e^{-\kappa\Delta t})}, \quad N = \mathcal{P}(\lambda), \quad \text{with } \mathcal{P} \text{ simulated by } \texttt{curand\_poisson}$$

and denoting $\mathcal{G}(\alpha)$ the standard gamma distribution whose simulation is presented in [8]

$$v_{t+\Delta t} = \frac{\sigma^2(1 - e^{-\kappa\Delta t})}{2\kappa} \mathcal{G}(d + N).$$

step 2. The integral $\int_0^1 v_s ds$ is stored in a variable `vI` set to zero before the for loop on time steps. Then, in the for loop we update `vI+=`$0.5 * (v_t + v_{t+\Delta t})\Delta t$.

step 3. Once we finish the for loop, we compute $\int_0^1 \sqrt{v_s} dW_s$, using the expression

$$\int_0^1 \sqrt{v_s} dW_s = \frac{1}{\sigma}\left(v_1 - v_0 - \kappa\theta + \kappa \texttt{vI}\right)$$

Then we compute

$$m = -0.5\texttt{vI} + \rho \int_0^1 \sqrt{v_s} dW_s, \quad \Sigma^2 = (1-\rho^2)\texttt{vI}$$

and we set $S_1 = \exp(m + \Sigma G)$ where G is a standard normal random variable independent from (G_1, G_2).

2. Define a device function that simulates the standard gamma distribution $\mathcal{G}(\alpha)$ presented in [8]. Make sure to deal with both situations $\alpha \geq 1$ and $\alpha < 1$. Then write down the code for the exact Monte Carlo simulation that is based on this standard gamma distribution. (5 points + 2 points)

3. For many values of $\kappa \in [0.1, 10]$, $\theta \in [0.01, 0.5]$ and $\sigma \in [0.1, 1]$ such that $20\kappa\theta > \sigma^2$, compare the execution time of the Euler discretization scheme to the almost exact scheme

$$\log S_{t+\Delta t} = \log S_t + k_0 + k_1 v_t + k_2 v_{t+\Delta t} + \sqrt{(1-\rho^2)v_t}\sqrt{\Delta t}(\rho G_1 + \sqrt{1-\rho^2} G_2)$$

with

$$k_0 = \left(-\frac{\rho}{\sigma}\kappa\theta\right)\Delta t$$

$$k_1 = \left(\frac{\rho\kappa}{\sigma} - 0.5\right)\Delta t - \frac{\rho}{\sigma}$$

$$k_2 = \frac{\rho}{\sigma}$$

presented in [10] where v_t is simulated as in the exact scheme. How the simulations are impacted when we choose $\Delta t = 1/30$ for the almost exact scheme? (3 points + 2 points)

2 Nested Monte Carlo for Log-Variance-Gamma model and neural network regression

In the Variance Gamma model, the asset price is modeled as the exponential of a Variance Gamma process. This Lévy process can be represented via Brownian subordination (Lévy density of the subordinator parametrized by κ). The variance Gamma process is described by the following dynamics:

$$X_{\mathrm{VG}}(t; \kappa, \sigma, \theta) := \theta Z_t + \sigma W(Z_t).$$

An asset Y is governed by the exponential VG dynamics if

$$Y_t = Y_0 e^{X_{\mathrm{VG}}(t)}.$$

There are three free parameters in total, they are κ, θ, and σ. The initial value of the asset price Y_0 is set to one.

The price of the call option is the expectation of the payoff in the risk-free measure:

$$C(T, K, \kappa, \theta, \sigma) = \mathbb{E}[(Y_T - K)^+].$$

In the two first questions, the students will be asked to generate a dataset of prices $\hat{C}$ obtained by Monte Carlo simulation such that $\hat{C}(T, K, \kappa, \theta, \sigma) = \hat{\mathbb{E}}[(Y_T - K)^+]$, where $\hat{\mathbb{E}}$ denotes the empirical expectation. The parameters are stored with the $\hat{C}$ and its confidence interval in the CSV files whose names contain strike and maturity. Second, the students will train a neural network $f(T, K, \kappa, \theta, \sigma) \approx \hat{C}(T, K, \kappa, \theta, \sigma)$ to approximate the Monte Carlo price. Last but not least, the students should study the monotonicity of f prices with respect to T and K and the convexity with respect to K.

Following [7], Y can be simulated by

$$Y_t = \exp\left(wt + \sum_{t_i \le t} \Delta X_{t_i}\right)$$

$$\Delta X_{t_i} = \sigma N_i \sqrt{\kappa \Delta S_i} + \theta \kappa \Delta S_i$$

$$\Delta S_i \sim \frac{x^{\Delta t/\kappa - 1}}{\Gamma(\Delta t/\kappa)} e^{-x} 1_{x>0}, \quad N_i \sim \mathcal{N}(0,1)$$

with $w = \ln\left(1 - \theta\kappa - \kappa\sigma^2/2\right)/\kappa$ being the adjustment term for martingale. The simulation can be performed using the following algorithms (from [5])

ALGORITHM 6.11 Simulating a variance gamma process on a fixed time grid

Simulation of $(X(t_1), \ldots, X(t_n))$ for fixed times $t_1, \ldots, t_n$: a discretized trajectory of the variance gamma process with parameters σ, θ, κ.

- *Simulate, using Algorithms 6.7 and 6.8, n independent gamma variables $\Delta S_1, \ldots, \Delta S_n$ with parameters $\frac{t_1}{\kappa}, \frac{t_2 - t_1}{\kappa}, \ldots, \frac{t_n - t_{n-1}}{\kappa}$.*
  Set $\Delta S_i = \kappa \Delta S_i$ for all i.
- *Simulate n i.i.d. $N(0,1)$ random variables $N_1, \ldots, N_n$.*
  Set $\Delta X_i = \sigma N_i \sqrt{\Delta S_i} + \theta \Delta S_i$ for all i.

The discretized trajectory is $X(t_i) = \sum_{k=1}^{i} \Delta X_i$.

ALGORITHM 6.7 Johnk's generator of gamma variables, $a \le 1$

REPEAT
    Generate i.i.d. uniform $[0,1]$ random variables U, V
    Set $X = U^{1/a}$, $Y = V^{1/(1-a)}$
UNTIL $X + Y \le 1$
Generate an exponential random variable E
RETURN $\frac{XE}{X+Y}$

ALGORITHM 6.8 Best's generator of gamma variables, $a \ge 1$

Set $b = a - 1$, $c = 3a - \frac{3}{4}$
REPEAT
    Generate i.i.d. uniform $[0,1]$ random variables U, V
    Set $W = U(1-U)$, $Y = \sqrt{\frac{c}{W}}(U - \frac{1}{2})$, $X = b + Y$
    If $X < 0$ go to REPEAT
    Set $Z = 64W^3V^3$
UNTIL $\log(Z) \le 2(b \log(\frac{X}{b}) - Y)$
RETURN X

1. Write down the device functions associated to algorithms 6.7, 6.8 and 6.11. (8 points)

2. Write down the Nested Monte Carlo simulation code that generates the required dataset (Get inspired by Exponential Ornstein–Uhlenbeck code). (7 points)

3. Train a neural network f that approximates the prices. Study the monotonicity of f with respect to T and K, as well as its convexity with respect to K and how can it be improved? (3.5 points + 1.5 points)

3 Nested Monte Carlo for Asian options and polynomial regression

Let F be the price of an Asian option whose value is given by $F(t,x,y) = E(X\big|S_t = x, I_t = y)$, $X = (S_T - I_T)_+$ where the processes I and S are defined by

$$I_t = \frac{1}{t}\int_0^t S_s ds, \quad dS_t = S_t \sigma dW_t \text{ and } S_0 = x_0.$$

- K, T are respectively the contract's strike and maturity; we assume $K = x_0 = 100$ and $T = 1$;
- W is a Brownian motion;
- $\sigma = 0.2$ is the volatility.

Both processes S and I are simulated using the following discretization scheme

$$S^i_{t_k} = S^i_{t_{k-1}}(1 + \sigma\sqrt{\Delta t} G^i_k) \text{ and } I^i_{t_k} = [t_{k-1} I^i_{t_{k-1}} + 0.5 * (S^i_{t_k} + S^i_{t_{k-1}})\Delta t]/t_k,$$

where $I_0 = 0$, $0 = t_0 < t_1 < ... < t_{100} = T = 1$, $t_k - t_{k-1} = \Delta t = 1/100$ and $\{G^i_k\}^{i=1,...,M}_{k=1,...,N}$ is a family of independent identically distributed random variables with standard normal distribution $\mathcal{N}(0,1)$. We assume $M = 256 * 256$.

1. In this part we use nested Monte Carlo for the simulation of the triplet $\{S^i_{t_k}, I^i_{t_k}, F(t_k, S^i_{t_k}, I^i_{t_k})\}^{i=1,...,M}_{k=1,...,N-1}$ and store the values on the GPU global memory (8 points)

   a) Based on the Monte Carlo developed during the course, implement a cuda program that simulates $\{S^i_{t_k}, I^i_{t_k}\}^{i=1,...,M}_{k=1,...,N-1}$ and store the values to the global memory. (4 points).

   b) Based on the nested Monte Carlo developed during the course, for the value of each couple $(S^i_{t_k}, I^i_{t_k})$, implement a cuda program that simulates $F(t_k, S^i_{t_k}, I^i_{t_k})$ and store the values to the global memory. (4 points).

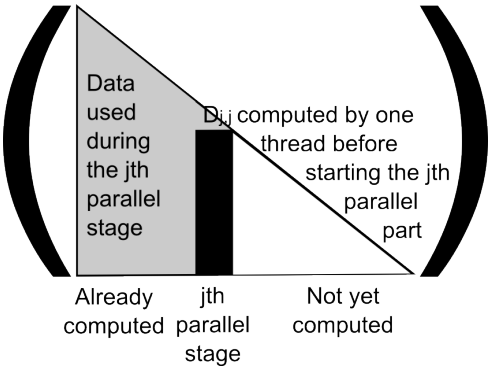
In the following, we want to regress the values of $F(t_{k+1}, S_{t_{k+1}}, I_{t_{k+1}})$ on monomials $\psi(S_{t_k}, I_{t_k})$. The regression, approximates the function $F(t_k, x, y)$ at each $k = 1, ..., N-1$ by $A_{t_k} \cdot \psi(x,y) = a^0_{t_k} + a^1_{t_k} x + a^2_{t_k} y + a^3_{t_k} x^2 + a^4_{t_k} y^2 + a^5_{t_k} xy$. In order to compute the regression coefficients $A_{t_k} = (a^0_{t_k}, a^1_{t_k}, a^2_{t_k}, a^3_{t_k}, a^4_{t_k}, a^5_{t_k})'$ with $'$ being the transpose operator, we need to use the trajectories drawn in question 1.a) to compute the following regression vector and matrix

$$\Gamma_{t_k} = \frac{1}{M}\sum_{i=1}^M \psi'(S^i_{t_k}, I^i_{t_k})\psi(S^i_{t_k}, I^i_{t_k}), \quad V_{t_k} = \frac{1}{M}\sum_{i=1}^M F(t_{k+1}, S^i_{t_{k+1}}, I^i_{t_{k+1}})\psi(S^i_{t_k}, I^i_{t_k}),$$

with $\psi(x,y) = (1, x, y, x^2, y^2, xy)'$.

2. Implement a cuda code that computes all the pairs $(\Gamma_{t_k}, V_{t_k})_{k=1,...,N-1}$ within a single kernel execution. (6 points)

The regression is performed through solving the linear systems $\Gamma_{t_k} A_{t_k} = V_{t_k}$ to get the values of $(A_{t_k})_{k=1,...,N-1}$. To solve these systems we need first to perform the LDLt factorization given to you of $(\Gamma_{t_k})_{k=1,...,N-1}$. As shown on the figure



the parallel implementation of LDLt (cf. [1]) is performed as follows: for a fixed value of j, the different coefficients $\{L_{i,j}\}_{j+1\leq i\leq d}$ can be computed by at most $d-j$ independent threads. Thus, $\{L_{i,1}\}_{2\leq i\leq d}$ involves the biggest number of possible independent threads equal to $d-1$. In this collaborative version, we use the maximum $d-1$ threads $+1$ additional thread that is involved in the copy from global to shared and in the solution of the system after factorization. This makes d threads for the collaborative version and one of these threads is also involved in the computation $D_{j,j}$ which needs a synchronization before calculating $L_{i,j}$.

3. Implement a cuda code that computes all the pairs $(A_{t_k})_{k=1,\dots,N-1}$ within a single kernel execution. Compare the values $A_{t_k}\cdot\psi(S^i_{t_k},I^i_{t_k})$ to $F(t_k,S^i_{t_k},I^i_{t_k})$ and estimate the regression error. (4 points + 2 points)

4 Merge big and batch sort small

The project begins with the implementation of the merge path algorithm introduced in [6], using a single CUDA block. Students are then required to extend this implementation to handle batched sorting of small arrays, followed by the merging of large arrays.

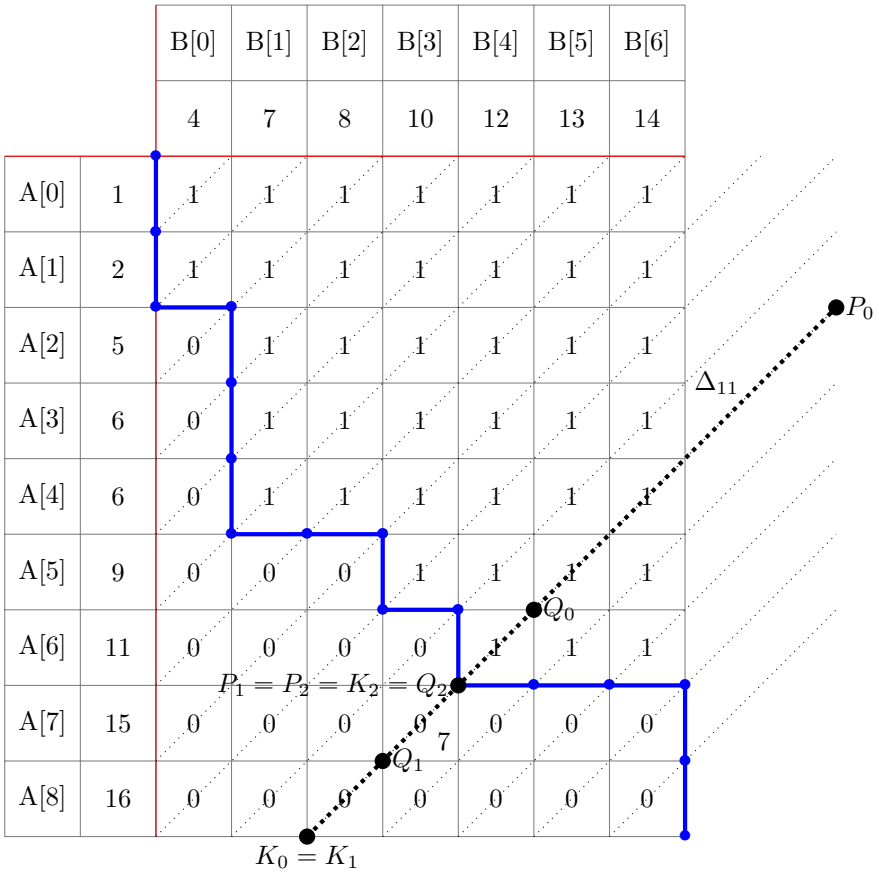
We start then with the merge path algorithm. Let A and B be two ordered arrays (increasing order), we want to merge them in an M sorted array. The merge of A and B is based on a path that starts at the top-left corner of the $|A|\times|B|$ grid and arrives at the down-right corner. The Sequential Merge Path is given by Algorithm 1 and an example is provided in the following figure.

Algorithm 1 Sequential Merge Path

Require: A and B are two sorted arrays
Ensure: M is the merged array of A and B with $|M|=|A|+|B|$
procedure MERGEPATH (A,B,M)
$j=0$ and $i=0$
while $i+j<|M|$ **do**
if $i\geq|A|$ **then**
M[i+j]=B[j]
$j=j+1$ ▷ The path goes right
else if $j\geq|B|$ or $A[i]<B[j]$ **then**
M[i+j]=A[i] ▷ The path goes down
$i=i+1$
else
M[i+j]=B[j]
$j=j+1$ ▷ The path goes right
end if
end while
end procedure

Algorithm 2 Merge Path (Indices of n threads are integers from 0 to $n-1$)

Require: A and B are two sorted arrays
Ensure: M is the merged array of A and B with $|M| = |A| + |B|$
  for each thread **in parallel do**
    i=index of the thread
    if $i > |A|$ **then**
      $K = (i - |A|, |A|)$ ▷ Low point of diagonal
      $P = (|A|, i - |A|)$ ▷ High point of diagonal
    else
      $K = (0, i)$
      $P = (i, 0)$
    end if
    while True **do**
      $offset = abs(K_y - P_y)/2$
      $Q = (K_x + offset, K_y - offset)$
      if $Q_y \geq 0$ and $Q_x \leq B$ and
        $(Q_y = |A|$ or $Q_x = 0$ or $A[Q_y] > B[Q_x - 1])$ **then**
        if $Q_x = |B|$ or $Q_y = 0$ or $A[Q_y - 1] \leq B[Q_x]$ **then**
          if $Q_y < |A|$ and $(Q_x = |B|$ or $A[Q_y] \leq B[Q_x])$ **then**
            $M[i] = A[Q_y]$ ▷ Merge in M
          else
            $M[i] = B[Q_x]$
          end if
          Break
        else
          $K = (Q_x + 1, Q_y - 1)$
        end if
      else
        $P = (Q_x - 1, Q_y + 1)$
      end if
    end while
  end for



Each point of the grid has a coordinate $(i, j) \in [\![0, |A|]\!] \times [\![0, |B|]\!]$. The merge path starts from the point $(i, j) = (0, 0)$ on the left top corner of the grid. If $A[i] < B[j]$ the path goes down else it goes right. The array $[\![0, |A|-1]\!] \times [\![0, |B|-1]\!]$ of boolean values $A[i] < B[j]$ is not important in the algorithm. However, it shows clearly that the merge path is a frontier between ones and zeros.

To parallelize the algorithm, the grid has to be extended to the maximum size equal to $\max(|A|, |B|) \times \max(|A|, |B|)$. We denote K_0 and P_0 respectively the low point and the high point of the ascending diagonals Δ_k. On GPU, each thread $k \in [\![0, |A| + |B| - 1]\!]$ is responsible of one diagonal. It finds the intersection of the merge path and the diagonal Δ_k with a binary search described in Algorithm 2.

1. For $|A| + |B| \leq 1024$, write a kernel `mergeSmall_k` that merges A and B using only one block of threads. (8 points)

In this part, we assume that we have a large number $N(\geq 1e3)$ of arrays $\{A_i\}_{1 \leq i \leq N}$ and $\{B_i\}_{1 \leq i \leq N}$ with $|A_i| + |B_i| = d \leq 1024$ for each i. Using some changes on `mergeSmall_k`, we would like to write `mergeSmallBatch_k` that merges two by two, for each i, A_i and B_i.

Given a fixed common size $d \leq 1024$, `mergeSmallBatch_k` is launched using the syntax

```
mergeSmallBatch_k<<<numBlocks, threadsPerBlock>>>(...);
```

with `threadsPerBlock` is multiple of d but smaller than 1024 and `numBlocks` is an arbitrary sufficiently big number.

2.a) After figuring out why the indices

```
int Qt = threadIdx.x/d;
int tidx = threadIdx.x - (Qt*d);
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```

are important in the definition of `mergeSmallBatch_k`, write the kernel `mergeSmallBatch_k` that batch merges two by two $\{A_i\}_{1 \leq i \leq N}$ and $\{B_i\}_{1 \leq i \leq N}$. Study the execution time with respect to d. (4 points)

2.b) Write the kernel `sortSmallBatch_k` that sorts many arrays $\{M_i\}_{1 \leq i \leq N}$ of size $d \leq 1024$. Study the execution time with respect to d. (4 points)

Actually, by using the index `idx = threadIdx.x + blockIdx.x * blockDim.x`, it is possible to replace the per-block thread index `threadIdx.x` used in question 1 to enable merging of large arrays across multiple blocks. However, this approach is suboptimal. Indeed, as mentioned in [6], merge path algorithm should be divided into 2 stages: partitioning stage and merging stage. The partitioning stage is important to propose an algorithm that involves various blocks.

3. For an input size $|A| + |B| = d$ large enough to fully engage the GPU's computational resources, design a parallel solution that merges arrays A and B by distributing the workload across multiple blocks through partitioning then study the execution time with respect to d. (4 points)

5 PDE simulation of bullet options

Assuming the Black & Scholes model, for time increments s and t with $0 \leq s < t \leq T$, the asset price process can be simulated using the following time induction formula:

$$S_t = S_s \exp\left((r - \sigma^2/2)(t-s) + \sigma\sqrt{t-s}G\right) \text{ and } S_0 = x_0 \text{ where}$$

- σ is the volatility assumed here equal to 0.2,
- r is the risk-free rate assumed here equal to 0.1,
- x_0 is the initial spot price of S at time 0, assumed here equal to 100

- G is independent from S_s and has a standard Normal distribution $\mathcal{N}(0,1)$.

This formula allows for simulating the path of the asset price from time s to t. This approach can be iteratively applied to generate an entire price path S for any time schedule $T_0 = 0 < T_1 < ... < T_M < T_{M+1} = T$.

We begin with a Monte Carlo simulation, in question 1, to approximate the price of a bullet option, which will serve as a reference for debugging the code in the subsequent questions. Questions 2 and 3 are then devoted to the PDE-based simulation of bullet options.

The price of a bullet option $F(t,x,j) = e^{-r(T-t)}E(X\big|S_t = x, I_t = j)$, $X = (S_T - K)_+ 1_{\{I_T \in [P_1,P_2]\}}$ with $I_t = \sum_{T_i \le t} 1_{\{S_{T_i} < B\}}$ and

- K is the contract's strike assumed here equal to $x_0 = 100$,
- T is the contract's maturity assumed here equal to 1
- barrier B should be bigger than S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$ where P_1 and P_2 are two integers.

Thus starting at any time $T_k \in \{T_0, T_1, ..., T_M\}$ with a starting value of $(S_{T_k}, I_{T_k}) = (x, j)$, one can compute $F(T_k, x, j) = e^{-r(T-T_k)}E(X\big|S_{T_k} = x, I_{T_k} = j)$ through the Monte Carlo simulation of the trajectories at $T_n > T_k$ by

$$S_{T_n} = S_{T_{n-1}} \exp\left((r - \sigma^2/2)(T_n - T_{n-1}) + \sigma\sqrt{T_n - T_{n-1}}G\right), \quad S_{T_k} = x,$$

$$I_{T_n} = I_{T_{n-1}} + 1_{\{S_{T_n} < B\}} \quad \text{and} \quad I_{T_k} = j.$$

1. Assuming $B = 120$, $M = 100$, $P1 = 10$, $P2 = 50$ and $T_i = i/M$ for $i = 0, ..., M$, write down a Monte Carlo simulation code to approximate $F(T_k, x, j)$. (8 points)

Let $u(t,x,j) = e^{r(T-t)}F(t, e^x, j)$ where F is the price of a bullet option as before. One can then show that, on any interval $t \in [T_{M-k}, T_{M-k+1})$, $k = M, ..., 0$, $u(t,x,j)$ is the solution of the PDE

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t,x,j) + \mu \frac{\partial u}{\partial x}(t,x,j) = -\frac{\partial u}{\partial t}(t,x,j)$$

$$\text{with: } \mu = r - \frac{\sigma^2}{2}$$

. The final and boundary conditions are:

$$\begin{aligned} \bullet\ u(T,x,j) &= \max(e^x - K, 0)\mathbb{1}_{\{j \in [P_1,P_2]\}} \text{ for any (x, j)} \\ \bullet\ u(t, \log[K/3], j) &= \texttt{pmin} = 0 \\ \bullet\ u(t, \log[3K], j) &= \texttt{pmax} = 2K \end{aligned}$$

In addition, because of the discontinuity of the process I at times T_{M-k}, we need to set

$$u_{T_{M-k}}(S_{T_{M-k}}, j) = \begin{cases} u_{T_{M-k}}(S_{T_{M-k}}, P_2)\mathbb{1}_{\{S_{T_{M-k}} \ge B\}} & \text{if } j = P_2 \\ u_{T_{M-k}}(S_{T_{M-k}}, P_k^1)\mathbb{1}_{\{S_{T_{M-k}} < B\}} & \text{if } j = P_k^1 - 1 \\ \begin{bmatrix} u_{T_{M-k}}(S_{T_{M-k}}, j)\mathbb{1}_{\{S_{T_{M-k}} \ge B\}} \\ +u_{T_{M-k}}(S_{T_{M-k}}, j+1)\mathbb{1}_{\{S_{T_{M-k}} < B\}} \end{bmatrix} & \text{if } j \in [P_k^1, P_2 - 1] \end{cases} \tag{6}$$

with $P_k^1 = max(P_1 - k, 0)$.

From now on we use notations $u_t(x,j) = u(t,x,j)$ and $u_{k,i} = u(t_k, x_i, j)$. Following Crank Nicolson scheme, we get

$$q_u u_{k,i+1} + q_m u_{k,i} + q_d u_{k,i-1} = p_u u_{k+1,i+1} + p_m u_{k+1,i} + p_d u_{k+1,i-1}$$

$$q_u = -\frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x}, \quad q_m = 1 + \frac{\sigma^2\Delta t}{2\Delta x^2}, \quad q_d = -\frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x}$$
$$p_u = \frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x}, \quad p_m = 1 - \frac{\sigma^2\Delta t}{2\Delta x^2}, \quad p_d = \frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x}$$

The figure below shows an example of how PDE's backward resolution algorithm (with M = 10, $P_1 = 3, P_2 = 8$) is deployed with time on the x-axis and the set of values of I_t in the ordinate.

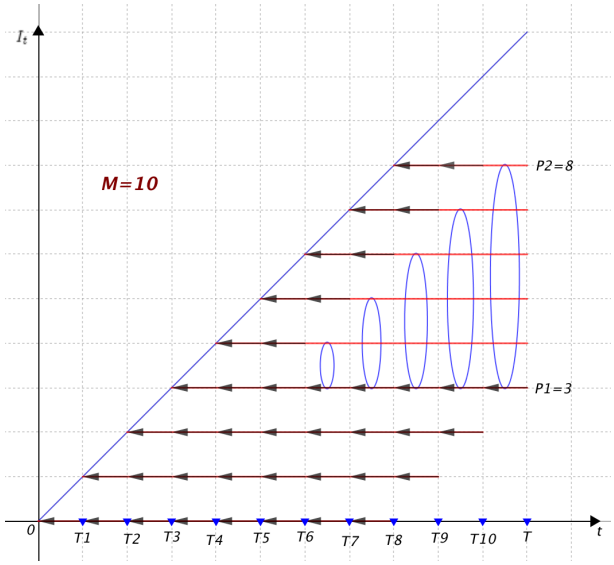


Figure 2: Backward induction scheme

2. Write the kernel that solves the PDE on $[T_M, T)$ and compare the numerical solution to the Monte Carlo values of $F(T_M, x, j)$. (4 points + 2 points)

3. After defining the kernel that performs (6), alternate between this kernel and the kernel of the previous question to solve all the PDE on $[T_0, T)$. Compare the numerical solution to the Monte Carlo values of $F(T_0, x, j)$. (4 points + 2 points)

6 Monte Carlo simulation of Hull-White model and sensitivities computation

The objective of this project is to implement Monte Carlo simulation for pricing and sensitivity analysis of fixed income instruments under the Hull-White one-factor short-rate model.

The Hull-White one-factor short-rate model is defined by:

$$dr(t) = \big(\theta(t) - ar(t)\big)\,dt + \sigma\,dW_t, \text{ and we assume } r(0) = 0.012,$$

where a is the mean reversion speed assumed equal to 1, $\sigma = 0.1$. In this project we consider $\theta(t)$ defined on the interval $[0, 10]$ by the following expression

$$\theta(t) = [0.012 + 0.0014 * t]1_{0\le t<5} + [0.019 + 0.001 * (t-5)]1_{5\le t\le 10}. \tag{7}$$

The distribution of $r(t)$ at t, knowing its value r_s at time $s < t$, is Gaussian and can be simulated by

$$r(t) = m_{s,t} + \Sigma_{s,t}G, \tag{8}$$

where G has a standard normal distribution and

$$m_{s,t} = r(s)e^{-a(t-s)} + \int_s^t e^{-a(t-u)}\theta(u)du, \quad \Sigma_{s,t} = \sqrt{\frac{\sigma^2(1 - e^{-2a(t-s)})}{2a}}.$$

The zero coupon bond price $P(t,T)$ represents the amount of money to pay at time t to obtain unity at T. Its value is given by

$$P(t,T) = E_t\left(\exp(-\int_t^T r_s ds),\right) \tag{9}$$

when $t=0$, $E_0 = E$ is a straight expectation. The forward rate $f(0,T)$ is expressed by

$$f(0,T) = \frac{\partial \ln(P(0,T))}{\partial T}.$$

1. Write a cuda program that, for a range of maturities of $T \in [0,10]$, computes Monte Carlo estimates of $P(0,T)$ and of $f(0,T)$. The integral $\int_0^T r_s ds$ should be approximated using the trapezoidal rule on a uniform time grid. (8 points)

In practice, the values of $\{P(0,T)\}_{T\in[0,10]}$ are quoted in the market and it is rather θ that is obtained using the following expression

$$\theta(T) = \frac{\partial f(0,T)}{\partial T} + af(0,T) + \frac{\sigma^2(1-e^{-2aT})}{2a}. \tag{10}$$

Also, under Hull-white, one can prove that the zero coupon defined in (9) has the following analytical expression (cf. [3])

$$P(t,T) = A(t,T)\,\exp\big(-B(t,T)r(t)\big), \text{ with } B(t,T) = \frac{1-e^{-a(T-t)}}{a} \text{ and}$$

$$A(t,T) = \frac{P(0,T)}{P(0,t)}\exp\left[B(t,T)f(0,t) - \frac{\sigma^2(1-e^{-2aT})}{4a}B(t,T)^2\right].$$

Consequently, with this analytical expression, one can simulate the trajectories of $\{P(t,T)\}_{0\le t\le T}$ through the simulation of $\{r_t\}_{0\le t\le T}$ and thus with a Monte Carlo simulation compute the value of the European call option

$$\mathbf{ZBC}(S_1,S_2,K) = E\left(e^{-\int_0^{S_1} r_s ds}(P(S_1,S_2)-K)_+\right), \text{ with } (x)_+ = \max(x,0) \text{ and } 0<S_1<S_2\le 10.$$

2. Use the values of $\{P(0,T)\}_{T\in\mathcal{T}_{10}}$ and of $\{f(0,T)\}_{T\in\mathcal{T}_{10}}$, with $\mathcal{T}_{10}$ being a uniform discretization of the interval $[0,10]$, computed in question 1

   a) to recover, from (10), the piecewise linear expression of θ given in (7) using a cuda kernel. (4 points)

   b) to perform a Monte Carlo simulation on GPU of $\mathbf{ZBC}(5,10,e^{-0.1})$. (4 points)

We call sensitivity of $\mathbf{ZBC}(S_1,S_2,K)$ with respect to the parameter σ its derivative expressed by

$$\partial_\sigma \mathbf{ZBC}(S_1,S_2,K) = E\left(\partial_\sigma P(S_1,S_2)e^{-\int_0^{S_1} r_s ds}1_{P(S_1,S_2)>K} - \left[\int_0^{S_1}\partial_\sigma r_s ds\right]e^{-\int_0^{S_1} r_s ds}(P(S_1,S_2)-K)_+\right),$$

and using the same G involved in (8), $\partial_\sigma r_t$ can be generated with the induction,

$$\partial_\sigma r(t) = M_{s,t} + \frac{\Sigma_{s,t}}{\sigma}G \text{ with } M_{s,t} = \partial_\sigma r(s)e^{-a(t-s)} + \frac{2\sigma e^{-at}[\cosh(at)-\cosh(as)]}{a^2} \text{ and } \partial_\sigma r(0) = 0.$$

3. We sssume the values of $\{P(0,T)\}_{T\in\mathcal{T}_{10}}$ and of $\{f(0,T)\}_{T\in\mathcal{T}_{10}}$, computed in question 1, as market data (fixed). Compute with Monte Carlo simulation on GPU the value of $\partial_\sigma \mathbf{ZBC}(5,10,e^{-0.1})$ and compare it to a finite difference approximation of the derivative at $\sigma = 0.1$. (3 points + 1 point)

7 Optimization using piecewise linear neural networks

We consider a neural network $G_L : \mathbb{R}^{d_0} \ni z_0 \mapsto G_L(z_0) \in \mathbb{R}$, defined through the iteration

$$\begin{cases} G_0(z_0) := z_0; \\ G_l(z_0) := \mathbf{a}(W_l G_{l-1}(z_0) + b_l),\ l = 1, \cdots, L-1; \\ G_L(z_0) := W_L G_{L-1}(z_0) + b_L. \end{cases} \tag{11}$$

The d_0 neurons in the input layer receive information from the environment and pass it onto the first hidden layer; for some positive integer L and $l = 1, \cdots, L-1$, each of the d_l neurons in the l-th hidden layer processes information received from the previous layer and passes it on to the next layer. The output is assumed here to be scalar.

Affine transformations are specified by weight matrices and bias vectors. For the l-th hidden layer, $l = 1, \cdots, L-1$ the weight is a $d_l \times d_{l-1}$ matrix W_l and the bias is a d_l-dimensional column vector b_l. With $d_L = 1$, the output layer has the weight as a d_{L-1}-dimensional row vector W_L and the bias as a real number b_L. We choose an activation function of the Leaky ReLU type, having the expression

$$a(z) := a_+ \max\{z, 0\} + a_- \min\{z, 0\}, \text{ for } z \in \mathbb{R}, \tag{12}$$

where the parameters a_+ and a_- are two positive real numbers. Although Leaky ReLU has $a_+ = 1$ and $a_- = 0.01$, we use the general expression (12) for its symmetry and genericity. Let the superscript T denote the transposition of a matrix or vector. For a d-dimensional real vector $u = (u_1, \cdots, u_d)^T$, the mapping $\mathbf{a}(u)$ is defined as

$$\mathbf{a}(u) := (a(u_1), \cdots, a(u_d))^T. \tag{13}$$

The input variables is supposed to belong to a d_0-polytope D_0 defined as

$$D_0 := \{x \in \mathbb{R}^{d_0} | C_D x \le \beta_D\}, \tag{14}$$

for an integer $m_0 > d_0$, a constant matrix $C_D \in \mathbb{R}^{m_0 \times d_0}$ and a constant vector $\beta_D \in \mathbb{R}^{m_0}$. The example considered here is $D_0 = [0, 1]^{d_0}$ with either $d_0 = 2$ or $d_0 = 3$.

One can show (cf. [2]) that the neural network G_L has the affine expression

$$G_L(z_0) = C_0(s_1, \cdots, s_{L-1}) z_0 - \beta_0(s_1, \cdots, s_{L-1}), \tag{15}$$

when z_0 is in the polytope

$$\mathcal{P}(s_1, \cdots, s_{L-1}) := \left\{ z_0 \in \mathbb{R}^{d_0} \middle| C(s_1, \cdots, s_{L-1}) z_0 \le \ \beta(s_1, \cdots, s_{L-1}) \right\}, \tag{16}$$

for any set of $s_1 \in \{-1, 1\}^{d_1}, \cdots, s_{L-1} \in \{-1, 1\}^{d_{L-1}}$. The collection

$$\mathscr{C} := \left\{ \mathcal{P}(s_1, \cdots, s_{L-1}) \middle| s_l \in \{-1, 1\}^{d_l},\ l = 1, \cdots, L-1 \right\} \tag{17}$$

is a finite partition of the polytope D_0, except for possible overlaps on the boundaries of the polytopes. The coefficients in (15) and (16) are constant for given values of $(s_1, \cdots, s_{L-1})$, and can be computed explicitly according to the following procedure.

(1) Initializing with

$$C_1(s_1) := -\operatorname{diag}(s_1) W_1 \text{ and } \beta_1(s_1) := \operatorname{diag}(s_1) b_1. \tag{18}$$

(2) If $L \ge 3$, for $l = 2, \cdots, L-1$, defining iteratively

$$\begin{cases} C_l(s_1, \cdots, s_l) := \operatorname{diag}(s_l) W_l \operatorname{diag}(\mathbf{a}(s_{l-1})) C_{l-1}(s_1, \cdots, s_{l-1}); \\ \beta_l(s_1, \cdots, s_l) := \operatorname{diag}(s_l) \left(W_l \operatorname{diag}(\mathbf{a}(s_{l-1})) \beta_{l-1}(s_1, \cdots, s_{l-1}) + b_l \right), \end{cases} \tag{19}$$

where $\operatorname{diag}(u)$ indicates the diagonal matrix with the vector u being its diagonal elements. Then the coefficients have the expressions

$$\begin{cases} C_0(s_1, \cdots, s_{L-1}) = -W_L \operatorname{diag}(\mathbf{a}(s_{L-1})) C_{L-1}(s_1, \cdots, s_{L-1}); \\ \beta_0(s_1, \cdots, s_{L-1}) = -\left(W_L \operatorname{diag}(\mathbf{a}(s_{L-1})) \beta_{L-1}(s_1, \cdots, s_{L-1}) + b_L \right), \end{cases} \tag{20}$$

$$C(s_1,\cdots,s_{L-1})=\begin{pmatrix} C_D \\ C_1(s_1) \\ \vdots \\ C_{L-1}(s_1,\cdots,s_{L-1}) \end{pmatrix} \text{ and } \beta(s_1,\cdots,s_{L-1})=\begin{pmatrix} \beta_D \\ \beta_1(s_1) \\ \vdots \\ \beta_{L-1}(s_1,\cdots,s_{L-1}) \end{pmatrix}. \tag{21}$$

Remark 1 *A vertex of a polytope is an extreme point, meaning a point that cannot be expressed as a nontrivial convex combination of other points in the polytope. Thus:*

- *any affine function defined on a polytope attains its extremal values at the vertices;*
- *a polytope without vertices is empty.*

The CUDA code provided, MC2V.cu for $d_0 = 2$ and MC3V.cu for $d_0 = 3$, does the following:

- reads the values of W and b from the file weights.txt then performs the affine decomposition (15) and (16) on polytopes in the kernel "Part_k";
- computes the vertices of each polytope calling the device function "Vertices",
- evaluates the affine expression on the vertices of each polytope calling the device function "levL"

The jupyter notebook, given to you, does the following:

- defines neural network architecture and trains neural networks to approximate functions;
- launches a Monte Carlo sampling of the maximum of the trained neural network;
- writes the values of W and b in the file weights.txt then compiles and executes either MC2V.cu or MC3V.cu to compute the maximum value through an evaluation on vertices.

Remark 2 *After running the Jupyter notebook with $d_0 = 2$ or $d_0 = 3$, and changing the target function accordingly, you should observe that*

- *the number of non-empty polytopes is much smaller than the total number of polytopes $nop = 4096$;*
- *the number of levels produced is smaller than the number of vertices.*

1. The device function "levL" currently parallelizes computations over all polytopes (both empty and non-empty) and returns the values of the affine function at all vertices. Instead, implement a kernel function "levL_k" that computes the maximum of the affine function evaluated over the vertices of only the non-empty polytopes. In this kernel, parallelization should be applied both across the non-empty polytopes and across the vertices of each non-empty polytope. (8 points)

We now aim to train neural networks with $L > 4$ by serializing the partitioning: we start with the first four hidden layers and then apply the induction (19) on the non-empty polytopes which yields a larger family of non-empty polytopes on which we apply (19) till enumerating all possible non-empty polytopes.

2. Write a new version of "Part_k" that does not call "levL" and allows handling networks with $L > 4$ by using the appropriate serial looping in the main function. Do not forget to apply (20) at the last step of this loop. (8 points)

3. Using the new versions of "Part_k" and "levL_k", train deep neural networks to approximate various sophisticated functions ($d_0 = 2$ or $d_0 = 3$) and determine their maxima. Analyze the execution time of the cuda part. (2 points + 2 points).

References

[1] L. A. Abbas-Turki and S. Graillat (2017). Resolution of a large number of small random symmetric linear systems in single precision arithmetic on GPUs. `https://hal.archives-ouvertes.fr/hal-01295549/`

[2] L. Abbas-Turki, B. Alexandrine and Q. Li (2023). Polynomial distribution of feedforward neural network output. `https://hal.science/hal-04217029/`

[3] D. Brigo and F. Mercurio (2006). *Interest rate models—Theory and practice: With smile, inflation and credit*. Springer Berlin Heidelberg.

[4] M. Broadie and Ö. Kaya. Exact simulation of stochastic volatility and other affine jump diffusion processes. *Operations research*, 54(2):217–231, 2006.

[5] R. Cont and P. Tankov (2003): *Financial modelling with jump processes*. Chapman and Hall/CRC.

[6] O. Green, R. McColl and D. A. Bader GPU Merge Path - A GPU Merging Algorithm. *26th ACM International Conference on Supercomputing (ICS)*, San Servolo Island, Venice, Italy, June 25-29, 2012.

[7] D. B. Madan, P. P. Carr and E. C. Chang (1998): The variance gamma process and option pricing. *Review of Finance*, 2(1), 79–105.

[8] G. Marsaglia and T. Wai-Wan. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software*, 26(3):363–372, 2000.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (2002): *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press.

[10] A. Van Haastrecht and A. Pelsser (2010) Efficient, almost exact simulation of the Heston stochastic volatility model. *International Journal of theoretical and applied finance*, 13(01), 1–43.

[11] Y. Zhang, J. Cohen and J. D. Owens (2010): Fast Tridiagonal Solvers on the GPU. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 127–136.