1 Schéma explicite, implicite et Crank-Nicolson

Nous nous intéressons à la simulation d'Équation aux Dérivées Partielles (EDP) paraboliques en utilisant plusieurs schémas. On cherche ainsi $F(t,x) : [0,T] \times \mathbb{R}^+ \to \mathbb{R}^+$ solution de l'EDP rétrograde suivante:

$$\frac{\partial F}{\partial t}(t,x) + rx\frac{\partial F}{\partial x}(t,x) + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 F}{\partial x^2}(t,x) = rF(t,x), \quad F(T,x) = f(x). \tag{1}$$

En utilisant le changement de variable $u(t,x) = e^{r(T-t)}F(t,e^x)$, il suffira de résoudre l'EDP équivalente

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t,x) + \mu\frac{\partial u}{\partial x}(t,x) = -\frac{\partial u}{\partial t}(t,x), \quad \mu = r - \frac{\sigma^2}{2}, \quad u(T,x) = f(e^x). \tag{2}$$

Dans la première partie, les étudiants doivent écrire la syntaxe de `PDE_diff_k1` et de `PDE_diff_k2` qui résolvent (2) de façon explicite pour plusieurs valeurs de σ. Dans la deuxième partie, les étudiants doivent écrire la syntaxe de `PDE_diff_k3` pour une résolution implicite et celle de `PDE_diff_k4` pour une résolution utilisant le schéma de Crank-Nicolson. Dans la dernière partie, en utilisant une grille de valeurs de $u(.,.)$ les étudiants doivent proposer une façon efficace pour trouver la valeur de x^0 et $sigma^0$ qui minimisent $|u(x^0,\sigma^0) - u^0|$ pour chaque u^0 donné en entrée.

Dans toutes les parties, la valeur finale $f(e^x) = \max(K - e^x, 0)$. Dans le fichier `PDE.cu` les conditions aux bords `pmin` et `pmax`, associées à f, sont aussi données. Les valeurs des paramètres sont: $T = 1$, $K = 100$, $S_0 = 100$, $r = 0.1$ et on lancera plusieurs simulations paramétrées par σ qui prend ses valeurs dans $[0.1, 0.5]$. La discrétisation utilise 256 cellules pour la variable x et 64 cellules pour σ. De plus, on discrétise $[0,T]$ avec 10000 pas de temps.

En considérant un schéma d'Euler explicite centré en espace, la simulation de u se fait grâce à:

$$\begin{gathered} u_{i,j} = p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1}, \quad u_{i,j} = u(t_i, x_j), \quad u(T,x) = f(e^x), \\ p_u = \frac{\sigma^2\Delta t}{2\Delta x^2} + \frac{\mu\Delta t}{2\Delta x}, \quad p_m = 1 - \frac{\sigma^2\Delta t}{\Delta x^2}, \quad p_d = \frac{\sigma^2\Delta t}{2\Delta x^2} - \frac{\mu\Delta t}{2\Delta x}. \end{gathered} \tag{3}$$

1. En gardant la boucle temporelle à l'extérieur, définir la syntaxe de `PDE_diff_k1` pour la simulation de (3) qui approxime bien la valeur de $F(0,x)$ lorsque x est compris dans l'intervalle $[0.7 * S_0, 1.3 * S_0]$.

2. En introduisant maintenant la boucle temporelle dans le kernel, définir la syntaxe de `PDE_diff_k2` pour la simulation de (3) qui approxime bien la valeur de $F(0,x)$ lorsque x est compris dans l'intervalle $[0.7 * S_0, 1.3 * S_0]$. Dans ce code, utiliser une allocation dynamique de la mémoire shared.

3. Comparer la solution 1 et 2 et déboguer le programme en utilisant la solution analytique $F(t,x)$ de l'EDP (1) donnée par la formule fermée:

$$F(t,x) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)x, \quad N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-y^2/2}dy. \tag{4}$$

avec $d_1 = \dfrac{\ln(x/K) + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}}$ et $d_2 = d_1 - \sigma\sqrt{T-t}$, N est la fonction de répartition d'une gaussienne calculée grâce à la fonction `NP` déjà donnée.

En considérant un schéma d'Euler implicite centré en espace, la simulation de u se fait grâce à:

$$u_{i+1,j} = q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1}, \quad u_{i,j} = u(t_i, x_j), \quad u(T,x) = f(e^x),$$
$$q_u = -\frac{\sigma^2 \Delta t}{2\Delta x^2} - \frac{\mu \Delta t}{2\Delta x}, \quad q_m = 1 + \frac{\sigma^2 \Delta t}{\Delta x^2}, \quad q_d = -\frac{\sigma^2 \Delta t}{2\Delta x^2} + \frac{\mu \Delta t}{2\Delta x} \tag{5}$$

3. En introduisant la boucle temporelle dans le kernel, définir la syntaxe de `PDE_diff_k3` pour la simulation de (5) qui approxime bien la valeur de $F(0,x)$ lorsque x est compris dans l'intervalle $[0.7 * S_0, 1.3 * S_0]$. Dans ce code, utiliser une allocation dynamique de la mémoire shared.

Avec un Schéma de Crank-Nicolson, on trouve la récurrence suivante

$$q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1} = p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1} \tag{6}$$

5. En introduisant la boucle temporelle dans le kernel, définir la syntaxe de `PDE_diff_k4` pour la simulation de (6) qui approxime bien la valeur de $F(0,x)$ lorsque x est compris dans l'intervalle $[0.7 * S_0, 1.3 * S_0]$. Dans ce code, utiliser une allocation dynamique de la mémoire shared.

Concernant la résolution des systèmes de tailles n fixé, un système par block, on utilise une réduction cyclique parallèle (parallel cyclic reduction) pour chaque système dans chaque block de threads. Cette routine est aussi donnée. Cette méthode consiste à résoudre un système $TX = Y$ où

$$T = \begin{pmatrix} d_1 & c_1 & & & & \\ a_2 & d_2 & c_2 & & 0 & \\ & a_3 & d_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & d_n \end{pmatrix}$$

et n est une puissance de 2 comme $n = 256, 512, ...$ en passant d'un système de n équations à n inconnues à deux systèmes de $n/2$ équations à $n/2$ inconnues, puis à 4 systèmes de $n/4$ équations à $n/4$ inconnues, ..., jusqu'à $n/2$ systèmes de 2 équations à 2 inconnues qui peuvent être facilement résolues.

Dans cette dernière partie, les étudiants doivent utiliser le tableau des solutions $u(.,.)$ obtenu par l'une des méthodes de simulation. Le but est d'exploiter à la fois la parallélisation et la monotonie marginale de $u(.,.)$ pour écrire un algorithme qui trouve la valeur de x^0 et $sigma^0$ en minimisant $|u(x^0, \sigma^0) - u^0|$ pour chaque u^0 donné en entrée.

6. Définir votre meilleure solution lorsqu'on donne en entrée une seule valeur u^0.

7. Définir votre meilleure solution lorsqu'on donne en entrée plusieurs valeurs $(u_i^0)_{i=1,...,M}$ auxquelles on associes en sortie $(x_i^0, sigma_i^0)_{i=1,...,M}$.

2 Batch LDLt factorization

Assuming a large number of $d \times d$ ($d \leq 1024$) symmetric and positive definite matrices $(A_n)_{1 \leq n \leq N}$, students have to optimize LDLt factorization to solve various linear systems associated to $(A_n)_{1 \leq n \leq N}$ at the same time.

For any kernel `myK` that batch computes problems of dimension d, we launch it using the syntax `myK<<<numBlocks, threadsPerBlock>>>(...);` with `threadsPerBlock` is multiple of d but smaller than 1024 and `numBlocks` is an arbitrary sufficiently big number. Explain why the indices

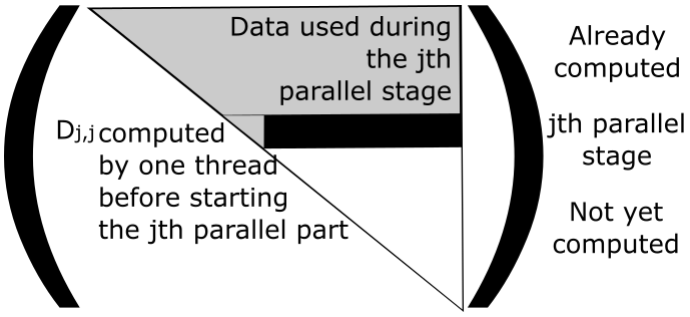
```
int tidx = threadIdx.x%d;
int Qt = (threadIdx.x-tidx)/d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```

are important in the definition of `myK`.

Let us start with $d \leq 64$. This exercise is the continuation of the LDLt exercise studied in the Chapter 2 of the lecture notes. Here, we develop another version that involves a bigger number of threads. Indeed, calling this new kernel is performed in the `main` function by

```
LDLt_max_k<<<NB,d*minTB,minTB*((d*d+d)/2+d)*sizeof(float)>>>(AGPU,
YGPU, d);
```

This new version requires `d` collaborative threads per linear system that perform a row after row computation. In fact, as shown on the figure below,



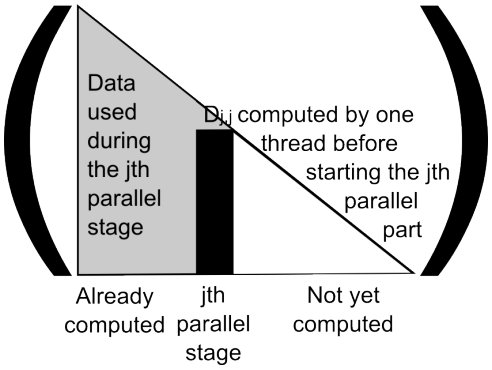
for a fixed value of j, the different coefficients $\{L_{i,j}\}_{j+1\leq i\leq d}$ can be computed by at most $d-j$ independent threads. Thus, $\{L_{i,1}\}_{2\leq i\leq d}$ involves the biggest number of possible independent threads equal to $d-1$. In this collaborative version, we use the maximum $d-1$ 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 $L_{i,j}$.

From the paragraph above make sure that you understand what is written in the following code

```
// Perform the LDLt factorization
 for(i=n; i>0; i--){
  if(tidx==0){
   for(k=n; k>i; k--){
    sA[nt+n2-i*(i+1)/2] -= sA[nt+n2-k*(k+1)/2]*
            sA[nt+n2-k*(k+1)/2+k-i]*
            sA[nt+n2-k*(k+1)/2+k-i];
   }
  }
  __syncthreads();
  if(tidx<i-1){
   sA[nt+n2-i*(i+1)/2+tidx+1] /= sA[nt+n2-i*(i+1)/2];
   for(k=n; k>i; k--){
    sA[nt+n2-i*(i+1)/2+tidx+1] -= sA[nt+n2-k*(k+1)/2]*
             sA[nt+n2-k*(k+1)/2+k-i]*
             sA[nt+n2-k*(k+1)/2+tidx+1+k-i]/
              sA[nt+n2-i*(i+1)/2];
   }
  }
  __syncthreads();
 }
```

Define this new kernel `LDLt_max_k`.

Now, write another kernel `LDLt_max_col_k` that performs the same computations as `LDLt_max_k` but on columns instead of rows as shown on the figure below.



For $d \leq 64$, compare the execution time of `LDLt_max_col_k` and of `LDLt_max_k`. For $64 < d \leq 1024$ explain and perform the needed adaptations then compare both kernels.

3 Merge large and batch merge small

This subject starts with the same algorithm of merge path, presented in [1], implemented on only one block. The students are then asked either to translate it into the merge of large arrays or the batch merge of small ones.

We start 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 Figure 1.

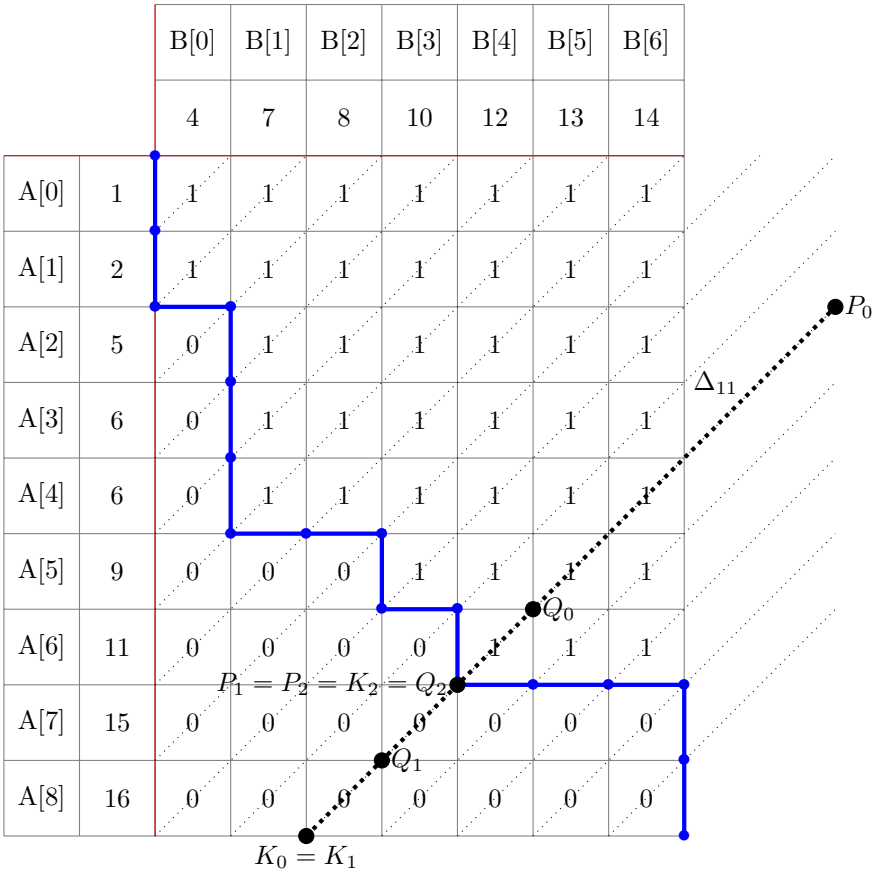


Figure 1: An example of Merge Path procedure

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 (Indexes of n threads are 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.

3.1 Merge large

As mentioned in [1], merge path algorithm is divided into 2 stages: partitioning stage and merging stage. The partitioning stage is important to propose an algorithm that involves various blocks.

2. For any size $|A|+|B|=d$ sufficiently smaller than the global memory, write a solution that merges A and B using various blocks.

3. Study the execution time with respect to $d = 4, 8, ..., 1024$.

3.2 Merge large and batch merge small

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.

4. Explain why the indices
```
int tidx = threadIdx.x%d;
int Qt = (threadIdx.x-tidx)/d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```
are important in the definition of `mergeSmallBatch_k`.

5. 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 $d = 4, 8, ..., 1024$.

4 PDE simulation of bullet options

The students have to implement and compare Thomas algorithm to PCR for tridiagonal systems. Then, they have to simulate a PDE of a bullet option using

Crank-Nicolson scheme based on either Thomas or PCR.

We refer to [3] for a fair description of PCR. Regarding Thomas algorithm, as described in [2], it allows to solve tridiagonal systems:

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{7}$$

using a forward phase

$$c'_1 = \frac{c_1}{b_1},\ y'_1 = \frac{y_1}{b_1},\ c'_i = \frac{c_i}{b_i - a_i c'_{i-1}},\ y'_i = \frac{y_i - a_i y'_{i-1}}{b_i - a_i c'_{i-1}} \text{ when } i = 2, ..., n \tag{8}$$

then a backward one

$$z_n = y'_n,\ z_i = y'_i - c'_i z_{i+1} \text{ when } i = n-1, ..., 1. \tag{9}$$

Using Thomas method, write a kernel that solves various tridiagonal systems ($d \leq 1024$) at the same time, one system per block. Do the same thing for PCR then compare both methods.

Let $u(t, x, j) = e^{r(T-t)} F(t, e^x, j)$ where F is 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 $I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$ and

- K, T are respectively the contract's strike and maturity
- $T_0 = 0 < T_1 < ... < T_M = T = T_{M+1}$ is a predetermined schedule
- barrier B should be bigger than S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r is the risk-free rate and σ is the volatility used in the Black & Scholes model

$$dS_t = S_t r dt + S_t \sigma dW_t, \quad S_0 = x_0.$$

$u(t, x, j)$ is then 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:

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

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

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

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

Noticing that:

$$\sum_{i=1}^{M} \mathbb{1}_{\{S_{T_i} < B\}} = \begin{cases} \sum_{i=1}^{M-1} \mathbb{1}_{\{S_{T_i} < B\}} & \text{if } S_{T_M} \geq B \\ \sum_{i=1}^{M-1} \mathbb{1}_{\{S_{T_i} < B\}} + 1 & \text{if } S_{T_M} < B. \end{cases}$$

Therefore we obtain the following backward induction:

for any $t \in [T_M, T[$, $\qquad u_t(x,j) = \mathbb{E}[(S_T - K)_+ | S_t = x]$

for any $t \in [T_{M-1}, T_M[$,

$$u_t(x,j) = \begin{cases} \mathbb{E}[(S_T - K)_+ \mathbb{1}_{\{S_{T_M} \geq B\}} \mid S_t = x] & \text{if } j = P_2 \\ \mathbb{E}[(S_T - K)_+ \mid S_t = x] & \text{if } j \in [P_1, P_2 - 1] \\ \mathbb{E}[(S_T - K)_+ \mathbb{1}_{\{S_{T_M} < B\}} \mid S_t = x] & \text{if } j = P_1 - 1 \end{cases}$$

for any $t \in [T_{M-k-1}, T_{M-k}[$, $k = M-1, ..., 1$,

$$u_t(x,j) = \begin{cases} \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_2) \mathbb{1}_{\{S_{T_{M-k}} \geq B\}} \mid S_t = x] & \text{if } j = P_2 \\ \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_k^1) \mathbb{1}_{\{S_{T_{M-k}} < B\}} \mid S_t = x] & \text{if } j = P_k^1 - 1 \\ \mathbb{E}\left[\begin{matrix} u_{T_{M-k}}(S_{T_{M-k}}, j) \mathbb{1}_{\{S_{T_{M-k}} \geq B\}} \\ + u_{T_{M-k}}(S_{T_{M-k}}, j+1) \mathbb{1}_{\{S_{T_{M-k}} < B\}} \end{matrix} \middle| S_t = x \right] & \text{if } j \in [P_k^1, P_2 - 1] \end{cases} \tag{10}$$

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

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. Write the pricing code and compare the execution time when using either Thomas or PCR.

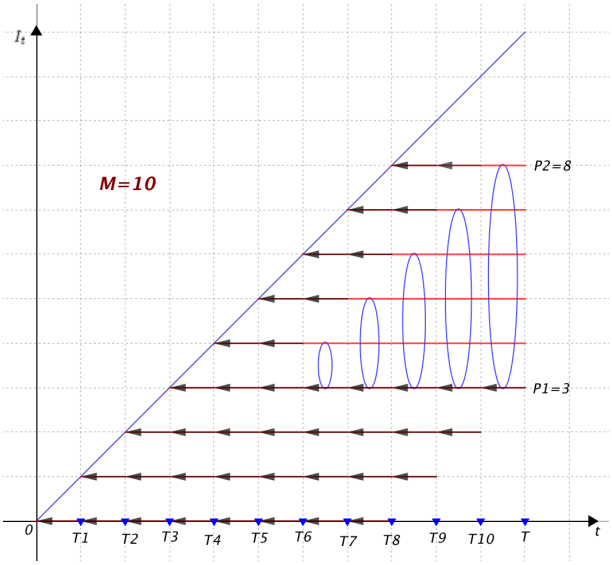


Figure 2: Backward induction scheme

5 Nested Monte Carlo vs. regression

Based on a nested Monte Carlo, the students have to simulate the price process $F(t,x,j)$ of a bullet option then compare the results to trained Neural Network (NN) or at least a linear regression on how to infer $F(t,x,j)$ for specific values taken by the triplet (t,x,j).

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 $I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$ and

- K, T are respectively the contract's strike and maturity
- $T_0 = 0 < T_1 < ... < T_M = T = T_{M+1}$ is a predetermined schedule
- barrier B should be bigger than S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r is the risk-free rate and σ is the volatility used in the Black & Scholes model

$$dS_t = S_t r dt + S_t \sigma dW_t, \quad S_0 = x_0.$$

The first step in this work is to make nested the Monte Carlo simulation developed in this course. Using nested Monte Carlo allows to simulate the value of $F(t,x,j)$ for various possible values of (t,x,j) instead of having only the simulation of $F(0,x_0,0)$ in a standard Monte Carlo.

1. Adapt the `rng.cu` and `rng.h` files to make it possible to simulate inner trajectories.
2. First simulate trajectories of $(S_t, I_t)_{t=0,T_1,...,T_M=T}$. On the top of these outer trajectories and starting at $(k,x,j) \in \{0,1,...,M\} \times \mathbb{R}_+ \times \{0,1,...,\min(k,P_2)\}$, write nested Monte Carlo code that allows to simulate realizations of $F(T_k,x,j)$.

In terms of accuracy and execution time, we want to compare the results provided by nested Monte Carlo to results obtained by the regression of $X = (S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}$ with respect to (S_t, I_t). Using Monte Carlo (not nested)

3. Write a linear regression solution that projects X on (S_{T_k}, I_{T_k}) in order to compute $F(T_k, x, j)$ then compare it to nested MC.

4. Train a NN capable of generating learned realizations of the price F and compare it to nested MC.

References

[1] 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.

[2] 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.

[3] 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.