Thème 3 : résolution numérique des équations aux dérivées partielles (EDP) linéaires

1 Résolution numérique de l'équation de Laplace à 2D

1.1 Formulation du problème 2D

On cherche à calculer numériquement le champ de température $T(x, y)$ à l'*intérieur* d'un domaine rectangulaire de côtés Δx et Δy (voir figure 1). En l'absence de source, ce champ vérifie l'équation de Laplace :

$$\Delta\, T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1}$$

Les quatre parois sont maintenues à des températures imposées (CL de type Dirichlet).

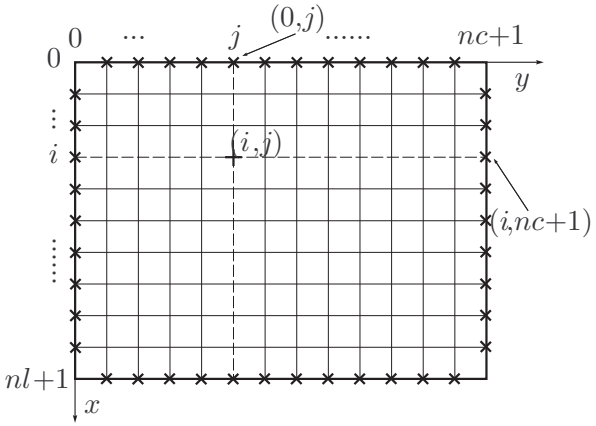

FIGURE 1 – Maillage du domaine : un point de la grille est repéré par un couple (i, j) où i représente le numéro de ligne et j le numéro de colonne. L'*intérieur* du domaine est numéroté de $i = 1$ à nl et de $j = 1$ à nc.

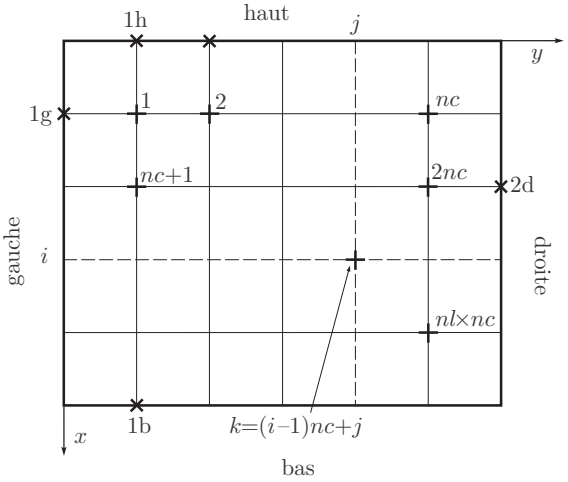

FIGURE 2 – Renumérotation des points intérieurs ligne par ligne de gauche à droite, puis de haut en bas (aplatissement RowMajor).

Le maillage choisi est une grille de pas $dx = dy = p$ représentée sur la figure 1. Chaque point est repéré par un couple d'indices (i, j) où le premier indice note le numéro de la ligne et le second celui de la colonne. On note $T_{i,j}$ la température en ce point, de coordonnées $x_i = i \times p$, $y_j = j \times p$. Les inconnues du problème sont les $n = nl \times nc$ valeurs de $T_{i,j}$ où $1 \leqslant i \leqslant nl, 1 \leqslant j \leqslant nc$. Elles seront déduites des valeurs de $T_{i,j}$ aux bords : nord ($i = 0$), sud ($i = nl + 1$), ouest ($j = 0$) ou est ($j = nc + 1$).

L'équation de Laplace discrétisée au second ordre se traduit par un système linéaire de n équations, chacune reliant la température au point (i, j) à celles des 4 points voisins :

$$T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4\,T_{i,j} = 0 \tag{2}$$

Les $2nl + 2nc$ températures des bords (aux points indiqués par des croix sur la figure 1) sont connues et les $n = nl \times nc$ températures intérieures sont à déterminer. Le pas p de la grille n'intervient pas dans ce calcul, mais sera utilisé dans la visualisation.

1.2 Renumérotation de la grille en 1D

N.-B. : Dans ce paragraphe, les indices sont les indices mathématiques commençant à 1. Penser à tenir compte du décalage d'une unité en C++.

On représente le champ de température à calculer par un tableau 1D noté Z à $n = nl \times nc$ composantes en renumérotant les points de la grille à l'intérieur du rectangle par un seul indice k en suivant les lignes (fig. 2). Il s'agit donc d'un aplatissement du domaine 2D en mode `RowMajor` :

$$\mathsf{Z}_k = T_{i,j} \quad \text{pour} \quad k = (i-1)nc + j \quad 1 \leqslant i \leqslant nl, 1 \leqslant j \leqslant nc \tag{3}$$

Pour un point sans contact avec la paroi ($1 < i < nl, 1 < j < nc$), l'équation (2) s'écrit :

$$\mathsf{Z}_{k-nc} + \mathsf{Z}_{k-1} - 4\mathsf{Z}_k + \mathsf{Z}_{k+1} + \mathsf{Z}_{k+nc} = 0 \tag{4}$$

Dès qu'il y a contact avec la paroi, l'opérateur Laplacien discrétisé fait intervenir une température imposée (ou au plus deux, dans les coins), ce qui va constituer le second membre B du système :

$$\mathsf{L}_{2D}\ \mathsf{Z} = \mathsf{B} \tag{5}$$

où L_{2D} est une matrice $n \times n$ constituée de $nl \times nl$ blocs carrés de dimension $nc \times nc$, du type :

$$\left(\begin{array}{cccc|cccc|cccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \\
\multicolumn{4}{c}{\longleftarrow nc \longrightarrow} & \multicolumn{4}{c}{\longleftarrow nc \longrightarrow} & \multicolumn{4}{c}{\longleftarrow nc \longrightarrow}
\end{array}\right)
\begin{array}{l}
\uparrow \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \downarrow
\end{array}
\quad \begin{array}{l} nl \text{ blocs} \\ \text{de taille} \\ nc \times nc \\ \\ \text{ici } nl = 3 \\ \text{et } nc = 4 \end{array} \tag{6}$$

et B est un vecteur à n composantes, dont au plus $2(nl+nc)-4$ non nulles, qui se calcule en sommant [1] les contributions des températures aux bords (voir représentation vectorielle dans le cours) :

$$\mathsf{B}_{k=(i-1)nc+j} = -\delta_{i,1}\, T_{i-1,j} - \delta_{i,nl}\, T_{i+1,j} - \delta_{j,1}\, T_{i,j-1} - \delta_{j,nc}\, T_{i,j+1} \tag{7}$$

1.3 Résolution du système à l'aide d'une première méthode de `Eigen` générique : `colPivHouseholderQr`

Créer un répertoire `te3`, puis deux sous répertoires de `te3` nommés `laplace` et `diffusion`. Copier dans votre répertoire `te3/laplace` le contenu du répertoire `laplace` disponible sur Moodle [2].

1.3.1 Programme principal : fichier `main.cpp`

Le programme principal devra lire les dimensions du domaine : `nl` et `nc`. Il déclarera ensuite :

- une `MatrixXf mat` (dont on ne précisera pas la taille à ce stade) où sera stockée la matrice L_{2D}
- pour stocker les conditions aux limites, deux `VectorXf ouest` et `est` de taille `nl` d'une part, deux `VectorXf nord` et `sud` de taille `nc` d'autre part
- deux `VectorXf sm` et `temp` (dont on ne précisera pas la taille à ce stade) où seront respectivement stockés le second membre B, et le champ des températures solution Z.

1. On rappelle la signification du symbole de Kronecker $\delta_{ij} = 1$ si $i = j$ et 0 sinon.
2. Le contenu de ce répertoire est également disponible sous `/data/MNCS/te3/laplace/` sur JupyterLab.

1.3.2 Construction de la matrice L_{2D} : fichier `matrices.cpp`

La fonction `matl2d` du module `matrices` remplira la matrice L_{2D} pour une grille `nl` × `nc`. Cette matrice est tridiagonale par blocs (carrés), et sera construite à partir de blocs élémentaires (bloc identité, bloc de la diagonale). Cette fonction respectera l'interface suivante :

```
MatrixXf matl2d(int nl, int nc);
```

Elle fera appel à la méthode `.block(i,j,p,q)` de `Eigen` qui permet de manipuler un bloc de taille `(p,q)` commençant à la position `(i,j)`.

Pour construire les blocs élémentaires, on définira deux autres fonctions d'interface :

```
MatrixXf mat_diag(int n);
MatrixXf mat_id(int n);
```

On veillera à construire le bloc élémentaire de la diagonale en une seule boucle.

Vérifier que la matrice L_{2D} a été correctement construite en l'affichant dans le programme principal, pour plusieurs valeurs (faibles) de `nl` et `nc` — en particulier en échangeant les dimensions.

1.3.3 Construction du second membre à partir des conditions aux limites : fichier `condlim.cpp`

Les conditions aux limites sont spécifiées sur les parois par quatre `VectorXf ouest` et `est nord` et `sud`. Ils seront calculés par la fonction `condlimites1`, qui respectera les interfaces suivantes :

```
void condlimites1(VectorXf &nord, VectorXf &sud, VectorXf &ouest, VectorXf &est);
```

Dans un premier temps, on suppose que chacune des parois est à une température uniforme (`tn`, `ts`, `tw` et `te`). Ces températures seront demandées à l'utilisateur dans `condlimites1`, et on pourra alors s'appuyer sur la méthode `VectorXf::Constant(...)` [3] pour l'initialisation des vecteurs.

Dans un second temps, on envisagera des gradients de température constants sur chaque paroi. La procédure `condlimites2` lira alors les températures aux deux extrémités de chaque paroi, soit 8 valeurs : `tnw` et `tsw` par exemple pour le bord ouest. Il suffira ensuite de saisir des extrêmes identiques pour retrouver une température uniforme.

Construction du second membre : Écrire une fonction `second_membre` qui construit le `VectorXf` creux B à `nl` × `nc` composantes du second membre à partir des `VectorXf` des températures sur les parois.

```
VectorXf second_membre(const VectorXf &nord, const VectorXf &sud,
                       const VectorXf &ouest, const VectorXf &est);
```

Cette fonction utilisera également une méthode d'initialisation de vecteur avancée, ainsi que les méthodes de manipulations des vecteurs par blocs [4].
Tester cette procédure en affichant le second membre.

1.3.4 Résolution du système avec `colPivHouseholderQr` de `Eigen`

Compléter le programme principal pour résoudre le système linéaire (5). On commencera par une méthode générale qui ne prend pas en compte les propriétés de la matrice L_{2D} : la fonction `colPivHouseholderQr` de la bibliothèque `Eigen`.

3. https://eigen.tuxfamily.org/dox/group__TutorialAdvancedInitialization.html
4. https://eigen.tuxfamily.org/dox/group__TutorialBlockOperations.html

1.3.5 Écriture du tableau des $T_{i,j}$ dans un fichier : `ecritemp`

S'il n'est pas intervenu dans la résolution du système, le pas p de la grille est nécessaire pour la visualisation en unités physiques. Les dimensions du domaine $\Delta x = p(nl+1)$ et $\Delta y = p(nc+1)$ étant fixées et le pas commun aux deux axes, nc et nl sont liés. p sera donc déduit par exemple de Δx et nl. Pour faciliter la vérification de la bonne construction de L_{2D} et B, la mise au point du code se fera avec $\Delta x = 4$, $\Delta y = 5$, $nl = 3$ et $nc = 4$, correspondant à une résolution $p = 1$. Par la suite, on fera varier la résolution p en modifiant le couple (nl, nc) de façon à couvrir toujours le même domaine, c'est-à-dire en conservant le rapport d'aspect $(nc + 1)/(nl + 1) = 5/4$.

Ajouter le calcul du pas et l'appel à la procédure `ecritemp` fournie (dans `io.cpp`) qui écrit le champ complet des températures (intérieur et bords) dans un fichier `laplace.dat` sous la forme d'un tableau 2D (`nl`+2) × (`nc`+2), précédé d'une ligne d'entête indiquant le nombre de lignes et de colonnes (`nl`, `nc`) du domaine intérieur et le pas `p`. Les températures aux 4 coins sont obtenues par moyennes des 2 voisins.

1.3.6 Visualisation sous python

Visualiser les isothermes et la surface $T(x, y)$ à l'aide du script python `plot-laplace.py` fourni. On pourra modifier le nombre d'isothermes via la variable `niso` et ajuster l'angle de vue de la surface via les variables `Altitude` et `Azimuth`.

1.3.7 Amélioration de la résolution et évaluation des performances

Reprendre le calcul et la visualisation avec `nl=19` et `nc=24` ($p = 0.2$) pour obtenir des isothermes plus régulières.

Évaluer le temps de calcul avec les instructions de chronométrage fournies dans le cours. Reprendre avec un pas 2 fois plus faible et noter les durées. Comparer le rapport des durées avec celui attendu. Quelle serait la durée de calcul nécessaire à la résolution du problème pour un pas $p = 0.05$? (Ne pas effectuer la résolution numérique correspondante.)

1.4 Amélioration des performances

Alors que la décomposition QR d'une matrice M ne requiert aucune propriété spécifique de la part de cette matrice (hormis qu'elle soit carré), il est beaucoup plus efficace de profiter des particularités de la matrice L_{2D} pour résoudre le système (5) en appelant des méthodes de résolution optimisées.

1.4.1 Décomposition de Cholesky avec `ldlt`

On choisit dans un premier temps d'exploiter le fait que la matrice L_{2D} est définie négative, en utilisant la fonction `ldlt` de la librairie Eigen.

Modifier le programme principal pour résoudre le système d'équation à l'aide de `ldlt`. Mesurer les performances de cette nouvelle méthode de résolution : comparaison avec `colPivHouseholderQr` pour $p = 0.2$ et $p = 0.1$, et évolution du temps de calcul de $p = 0.2$ à $p = 0.1$.
Quelle serait la durée de calcul nécessaire à la résolution du problème pour $p = 0.05$ avec `ldlt` ? (Ne pas effectuer la résolution numérique correspondante.)

1.4.2 Utilisation du caractère creux de L_{2D}

Lorsque le pas p de résolution diminue, la matrice L_{2D} devient de plus en plus creuse. Les calculs sur les matrices « denses », qui utilisent l'ensemble des éléments de la matrice (donc beaucoup de 0), deviennent de plus en plus inefficaces. Il devient ainsi intéressant d'utiliser les méthodes développées pour les matrices creuses : stockage des éléments et opérations sur ces matrices. Ces méthodes

nécessitent l'utilisation de l'entête `<Eigen/Sparse>`. Leur mise en œuvre est un peu plus complexe que pour les matrices denses, et est proposée à titre d'ouverture.

Création de la matrice creuse L_{2D}^{C}

La première étape consiste à créer une matrice creuse (nommée par exemple `matcr`) qui ne va contenir que les éléments non-nuls de L_{2D}. On s'appuiera sur **le deuxième exemple** fourni dans la documentation `Eigen`[5], dans le paragraphe « **Filling a sparse matrix** ».

On fera attention à bien définir la taille de `matcr`, ainsi qu'à réserver l'espace de stockage correspondant au nombre maximal d'éléments non-nuls par colonne de L_{2D}.

Chronométrer le temps nécessaire à la création de la matrice creuse.

Résolution à l'aide de la méthode `SimplicialLDLT`

Pour résoudre le système, on s'appuie sur l'exemple générique « **Sparse solver concept** »de la documentation `Eigen`[6].

Utiliser le `SolverClassName` nommé `SimplicialLDLT`, qui met en œuvre la décomposition de Cholesky pour les matrices creuses.

Chronométrer le temps de résolution pour un pas $p = 0.1$. Comparer avec les méthodes précédentes. Est-il envisageable de faire les calculs pour un pas plus petit ?

5. https://eigen.tuxfamily.org/dox/group__TutorialSparse.html
6. https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html

2 Résolution numérique de l'équation de diffusion 1D

2.1 Rappels introductifs

On cherche à obtenir numériquement la fonction $u(x,t)$ pour $t \in [0,T]$, sur l'intervalle $x \in [0,L]$, solution de l'équation de diffusion à 1D avec des conditions initiales (CI) et aux limites (CL) :

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \qquad \text{avec} \qquad \begin{cases} \text{(CI)} & \forall x,\ u(x,0) = u_0(x) \\ \text{(CL)} & \forall t,\ u(0,t) = u_g(t) \quad \text{et} \quad u(L,t) = u_d(t) \end{cases} \tag{8}$$

Le coefficient de diffusion D, nécessairement positif, est supposé constant pour simplifier.

L'intervalle $[0,L]$ est divisé en n_x+1 pas de largeur $\delta x = L/(n_x+1)$, et celui de temps en n_t+1 pas de durée δt (voir fig. 3, p. 8). On note u_j^n, la valeur numérique de $u(x,t)$ au point $x = j\delta x$ et au temps $t = n\delta t$. On définit le paramètre caractéristique :

$$\alpha = \frac{D\ \delta t}{(\delta x)^2} \tag{9}$$

— Les conditions aux limites fixent donc les u_0^n et les $u_{n_x+1}^n$ pour tout n ;
— La condition initiale donne u_j^0 pour $0 \leqslant j \leqslant n_x+1$.

La méthode de résolution est itérative en temps : à chaque pas d'intégration, on doit calculer les n_x valeurs aux points *intérieurs* du domaine en prenant en compte les conditions aux limites.

2.2 Partie commune

2.2.1 Méthodes et notations

On définit :

— le vecteur de taille n_x des u_j^n pour $j \in [1,n_x]$, $\mathsf{U}^n = (u_1^n, u_2^n \dots u_{n_x}^n)$;
— le vecteur de taille n_x déduit des conditions aux limites $\mathsf{V}^n = (u_g^n, 0 \dots, 0, u_d^n)$, supposé ici constant pour simplifier ;
— la matrice, de taille $n_x \times n_x$, $\mathsf{M}(\alpha) = \mathbb{1} - \alpha \mathsf{L}_{1D}$, où $\mathbb{1}$ est l'identité et L_{1D} la matrice tridiagonale, qui représente la dérivée seconde en différences finies à 3 termes.

$$\mathsf{L}_1 D = \underbrace{\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}}_{n_x} \Bigg\} n_x$$

Les différents algorithmes vus en cours s'écrivent :

Explicite : $\mathsf{U}^{n+1} = \mathsf{M}(-\alpha) \cdot \mathsf{U}^n + \alpha \mathsf{V}^n$
soit $u_j^{n+1} = \alpha\, u_{j-1}^n + (1-2\alpha)\, u_j^n + \alpha\, u_{j+1}^n$ pour $2 \leqslant j \leqslant n_x - 1$
plus les contacts avec les bords pour $j = 1$ et $j = n_x$;

Implicite : $\mathsf{U}^{n+1} = \mathsf{M}(\alpha)^{-1} \cdot \left[\mathsf{U}^n + \alpha \mathsf{V}^{n+1}\right]$;

Crank–Nicolson : $\mathsf{U}^{n+1} = \mathsf{M}(\frac{\alpha}{2})^{-1} \cdot \left[\mathsf{M}(-\frac{\alpha}{2}) \cdot \mathsf{U}^n + \frac{\alpha}{2}(\mathsf{V}^n + \mathsf{V}^{n+1})\right]$.

On rappelle que l'algorithme explicite n'est stable que pour $\alpha \leqslant 1/2$. On notera enfin que pour α petit et fini [7], la matrice $\mathsf{M}(\alpha)^{-1}$ contient des termes non-diagonaux qui décroissent comme $\alpha^{|i-j|}$.

2.2.2 Organisation des codes

On souhaite écrire un code modulaire permettant de changer aisément de méthode. Les différents fichiers sont :

7. pour $\alpha \ll 1$, $\mathsf{M}(-\alpha) \approx \mathsf{M}(\alpha)^{-1}$.

— `cond_init.cpp` pour le traitement des conditions initiales ;
— `matrices.cpp` pour la construction de la matrice $\mathsf{M}(\alpha)$ (sauf pour la méthode explicite) ;
— `methodes.cpp` pour les trois procédures d'avancement d'un pas en temps ;
— `util.cpp` pour l'écriture de l'entête et des u_j^n dans le fichier de résultat.

Pour ne pas avoir à stocker les données sous forme 2D, on choisit ici d'écrire dans un fichier les résultats des u_j^n au fur et à mesure de leur obtention. Cela permet aussi de conserver le début de la solution en cas de divergence. En vue du graphique, les valeurs aux bords seront aussi écrites dans le fichier à chaque pas de temps. On notera que :

— La matrice utilisée par la méthode explicite est tridiagonale : son action devra être codée «à la main» dans `methodes.cpp`, pour éviter de faire intervenir tous les éléments nuls.
— Au contraire, les produits matrice-matrice et/ou matrice-vecteur utilisés dans les méthodes implicite et Crank-Nicolson, impliquent des matrices denses, et seront réalisés à l'aide de l'opérateur `*` surchagé de Eigen.
— La matrice à inverser dans ces deux derniers schémas est constante, et ne nécessite qu'une seule inversion, qui sera réalisée par la fonction membre `.inverse()` de Eigen.

2.2.3 Spécifications du problème et choix des paramètres

On étudie la diffusion d'une superposition de deux modes sinusoïdaux :

— avec comme CI :

$$u_0(x) = 1 + \sin\left(\frac{\pi\, x}{L}\right) + 0.2\, \sin\left(\frac{5\,\pi\, x}{L}\right) \tag{10}$$

— et des CL constantes $u_g^n = u_0(x=0) = 1$ et $u_d^n = u_0(x=L) = 1$ quel que soit n.

Pour les tests, on choisira par défaut les paramètres suivants :

— $L = 12.0$ et une discrétisation fixe en espace avec $\delta x = 0.1$;
— $D = 1 \times 10^{-2}$: choisir le pas en temps δt équivaut donc à choisir α.

2.3 Travail à effectuer

2.3.1 Algorithme explicite

Créer un répertoire `te3/diffusion`. Y copier le contenu du répertoire `diffusion` sur Moodle[8].

Programme principal `main.cpp` chargé de :

— Lire les paramètres `dt` et `nt`.
— Sachant que `L` et `dx` sont fixés, calculer `nx` et α puis les afficher.
— Initialiser (sans les demander à l'utilisateur) `ug` et `ud`.
— Allouer les vecteurs `unew`, `uold` de taille `nx` et les matrices nécessaires[9] ;
— Initialiser `uold` avec les conditions initiales (10).
On écrira pour cela une fonction `VectorXf init2sinus(...)` dans `cond_init.cpp`.
— Écrire une fonction `void ecrit_entete(...)` dans `util.cpp` qui écrit, dans un fichier dont le nom est passé en argument, la ligne d'entête commençant par le caractère `#` et indiquant, dans l'ordre : le nombre de points en x ($n_x + 2$ avec les bords), n_t, δx, δt et α.
Écrire, dans le même fichier, une autre fonction `void ecrit(...)` pour écrire les u_n^j (bord compris).
Poursuivre le programme principal pour écrire la ligne d'entête et les conditions initiales dans `explicite.dat` (on utilisera par la suite `implicite.dat` et `cranknic.dat`)
— Effectuer la boucle sur les temps qui, à chaque pas :
1. appelle `avance_expl` pour calculer `unew` en fonction de `uold` ;
2. recopie `unew` dans `uold` ;
3. écrit les $n_x + 2$ valeurs obtenues, bords compris, dans le fichier `explicite.dat`

8. Le contenu de ce répertoire est également disponible sous `/data/MNCS/te3/diffusion/` sur JupyterLab.
9. On utilisera des objets `VectorXf` et `MatrixXf` de Eigen.

Méthode Écrire la procédure `avance_expl` implémentant **sans matrice** l'algorithme explicite.

Tests
- Tester le programme avec, pour commencer `nt=1000`, `dt=0.1`, `ug=1.`, `ud=1.`
- Visualiser le résultat : figure `explicite.pdf`.
- Augmenter `nt` pour comparer la constante de temps d'atténuation des deux modes ; commenter.
- Augmenter enfin la valeur de `dt` et observer ; commenter ce qui se passe si $\alpha > 0.5$.
  Dans ce cas, on arrêtera le calcul dans le programme principal dès qu'un des points de la solution dépasse un certain seuil.

2.3.2 Algorithme implicite

Codes
- Commenter la partie de `main.cpp` spécifique à la méthode explicite.
- Écrire, dans `matrices.cpp`, la fonction `MatrixXf matalpha(...)` qui construit $\mathsf{M}(\alpha)$.
- Poursuivre `main.cpp` pour déclarer et calculer les matrices `malpha` et `invmalpha`.
- Implémenter dans `methodes.cpp` la procédure `avance_impl` qui calcule `unew`.

Tests
- Tester l'implémentation réalisée avec `nt=1000`, `dt=0.1`.
- Visualiser le résultat : figure `implicite.pdf`. Puis augmenter `nt`
- Augmenter la valeur de `dt` et expliquer ce qui passe pour par exemple $\alpha = 1$.
- Étudier le cas d'une condition initiale en forme de marche : $u_0(x) = +1$ pour $0 \leqslant x \leqslant L/2$ et $u_0(x) = -1$ pour $L/2 < x \leqslant L$, avec des conditions aux limites constantes $u_g = +1$ et $u_d = -1$.

2.3.3 Algorithme de Crank–Nicolson

Codes
- Commenter la partie de `main.cpp` spécifique à la méthode implicite.
- Déclarer et calculer les matrices nécessaires à l'aide des procédures de `matrices.cpp`
- Implémenter dans `methodes.cpp` la procédure `avance_cranknic` qui calcule `unew` (on utilisera un `VectorXf` local pour stocker les vecteurs résultats intermédiaires).

Tests
- Tester l'implémentation réalisée avec `nt=1000`, `dt=0.1`.
- Visualiser le résultat : figure `cranknic.pdf`.
- Augmenter la valeur de `dt` pour tester le cas $\alpha > 1$.
- Comparer avec les résultats de l'algorithme implicite.

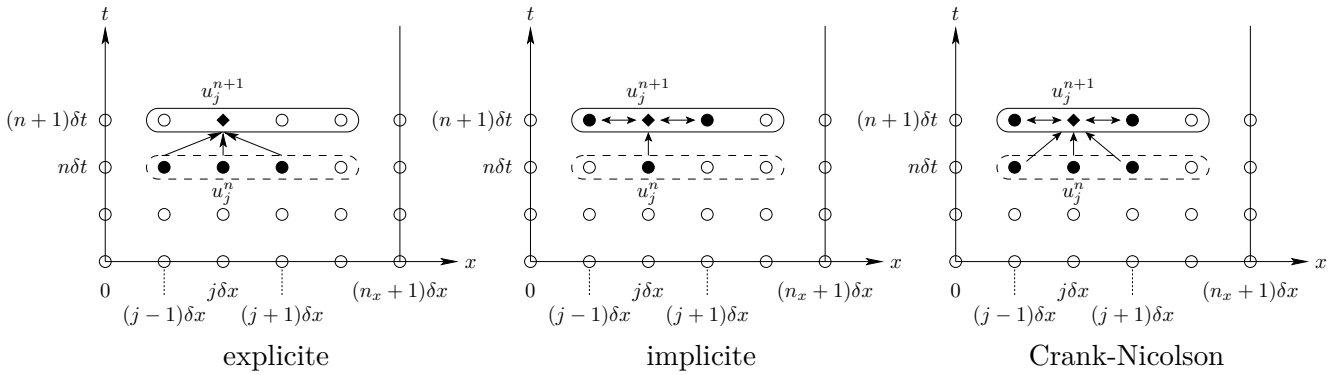

FIGURE 3 – Schémas différentiels des trois algorithmes