Mise en œuvre dans le cas $3D$: simulations *MPCube*.

Introduction.

Ce chapitre est consacré à la chaîne de calcul multi-échelle *SALOME-MPCube*, développée dans le cadre de cette thèse. Cette chaîne regroupe tous les outils nécessaires à la réalisation de simulations numériques par les méthodes Q_1/*VFDiam* et GD/*VFDiam*. Une description précise de ces deux méthodes est disponible au chapitre §3. La chaîne de calcul utilise principalement deux outils : la plate-forme *SALOME* [163] et le code de calcul *MPCube* [61].

La plate-forme *SALOME*, présentée à la section §6.1, intervient tout d'abord au début de la chaîne de calcul, afin de construire les maillages nécessaires aux futurs calcul *MPCube*. Cette partie de la chaîne de calcul est l'objet du chapitre §6.

Le code *MPCube*, basé sur le cœur de calcul *Trio-U* [171], intervient dans un second temps. Il résout les problèmes locaux, puis assemble et résout le problème grossier. Ces fonctionnalités, liées aux méthodes multi-échelles, n'existent pas dans le code originel de *MPCube*. Elles ont donc fait l'objet de développements spécifiques.

En fin de chaîne, *SALOME* intervient à nouveau, par le biais de ses outils MED-SPLITTER et VISU. La plate-forme permet d'effectuer des opérations de postraitement sur les champs solutions : calcul de la solution fine $C_{H,h}$, extraction et fusion des champs solutions, etc.

La chaîne de calcul multi-échelle *SALOME-MPCube* a été utilisée pour résoudre différents problèmes de diffusion au sein de matériaux cimentaires, que l'on présente au chapitre §7.

Dans un premier temps, on décrit les étapes de la chaîne de calcul multi-échelle *SALOME-MPCube*. On s'attache en particulier à présenter les possibilités de parallélisme étape par étape.

La seconde partie du chapitre est consacrée aux détails techniques de l'implémentation dans le code *MPCube* de la chaîne de calcul. On y présente tout d'abord les codes de calculs *Trio-U* et *MPCube* (§5.2.1) avant de détailler les différents objets ajoutés au code *MPCube* originel par la méthode multi-échelle (§5.2.2 et §5.2.3). On présente en outre une série de simulations ayant permis de qualifier l'implémentation de ces objets.

5.1 La chaîne de calcul multi-échelle *SALOME-MPCube*.

On peut espérer disposer de deux types de parallélisme au sein de la chaîne de calcul multi-échelle *SALOME-MPCube*.

Il y a tout d'abord le parallélisme intrinsèque aux méthodes multi-échelles, dit *extra*-cellulaire. La grande majorité du travail à effectuer sur les cellules du domaine, par exemple leurs discrétisations et la résolution des problèmes locaux, peut en effet s'effectuer indépendamment d'une cellule à l'autre. On peut donc traiter plusieurs cellules de front.

Il est également possible d'utiliser un second type de parallélisme, dit *intra*-cellulaire. Il s'agit de profiter des fonctionnalités de *SALOME* et *MPCube* qui permettent d'assigner plusieurs processeurs à une unique tâche. *MPCube* dispose en effet de solveurs parallèles, qui permettent de répartir l'effort de calculs de gros et très gros problèmes sur plusieurs cœurs de calculs. De même, *SALOME* peut faire appel à des mailleurs parallèles.

La Figure 5.1 présente un organigramme simplifié de la chaîne de calcul *SALOME-MPCube*. Sont également représentés les degrés de parallélisme, théoriques et implémentés, disponibles pour chaque module de l'organigramme.

Il est à noter que les étapes les plus coûteuses ont été parallélisées : résolution des problèmes locaux, construction des fonctions de bases et calcul des matrices associées. Comme on le détaille un peu plus bas, la génération des maillages est partiellement parallélisée dans la mesure où on construit chaque maillage de cellule indépendamment des autres. Pour des raisons techniques, cette construction s'effectue cependant séquentiellement, c'est-à-dire une cellule après l'autre.

5.1.1 Génération des maillages.

La première étape de la chaîne de calcul *SALOME-MPCube* consiste à discrétiser l'ensemble des cellules du découpage multi-échelle. Plus précisement, il s'agit, partant d'une description de la géométrie du domaine, d'obtenir un jeu de maillages exploitables par le code de calcul *MPCube*. Cette tâche est dévolue à la plate-forme *SALOME* et le chapitre §6 décrit les fonctionnalités spécifiquement développées dans ce but.

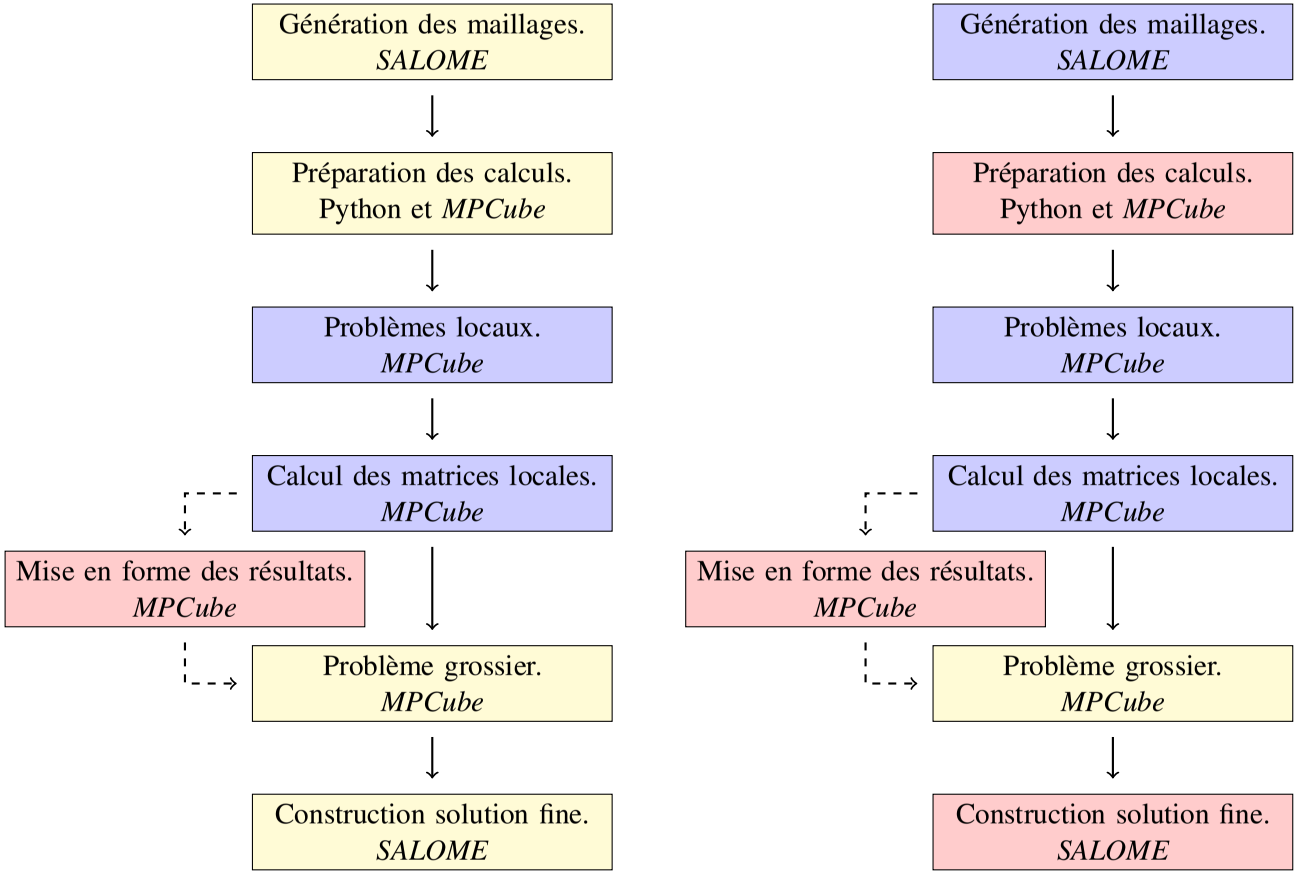

FIGURE 5.1 – Organigramme de la chaîne de calcul *SALOME-MPCube*. Les modules sont colorés en fonction du degré de parallélisme implémenté (à gauche) et théoriquement atteignable (à droite) : parallélismes *intra* et *extra*-cellulaire (en bleu), parallélisme *extra*-cellulaire seul (en rouge) et tâche séquentielle (en jaune).

Du point de vue du parallélisme, la génération de maillages dispose d'un parallélisme *extra*-cellulaire presque total. En effet, une fois les informations géométriques attribuées aux cellules, c'est-à-dire une fois les inclusions réparties (cf. Alg. 6.7 L.1), les travaux de construction géométrique et de discrétisation sont indépendants d'une cellule à l'autre. Dans le cas de la construction de maillages coïncidents pour la méthode GD/*VFDiam*, décrite à la section §6.3, il faut cependant tenir compte de la construction du maillage de référence (cf. Alg. 6.9 L.2), une tâche qui ne peut se faire que séquentiellement.

L'implémentation de ce module a été réalisée de manière à profiter de ce parallélisme. Ainsi, la fonction discrétisant une cellule ne requiert que la liste des inclusions correspondantes en entrée, et est appelée une fois par cellule. L'implémentation est donc indépendante d'une cellule à l'autre. Malgré cela, en pratique la discrétisation des cellules se fait séquentiellement. En effet, travailler de front sur plusieurs cellules demande de manipuler plusieurs instances de *SALOME*, ce qui est techniquement complexe car de nombreuses ressources sont partagées. Par le biais du module YACS, il est théoriquement possible de lancer une seule instance de *SALOME* qui piloterait les constructions paralléles. Cependant, cette option n'a pas été mise en place, le temps manquant pour maîtriser ce module.

La génération des maillages dispose d'un parallélisme *intra*-cellulaire. En effet, si la construction géométrique de la cellule ne peut que se faire séquentiellement, il n'en est pas de même de la discrétisation du domaine. *SALOME* dispose en effet d'un mailleur parallèle GHS3D PARALLEL [103] pouvant tirer partie de la disponibilité de plusieurs processeurs.

L'utilité de cette option reste cependant confinée à la création de très gros maillages, dépassant la dizaine de millions de volumes. Au contraire, le principe premier des méthodes multi-échelles est de travailler sur une série de problèmes de tailles raisonnables. Cette forme de parallélisme n'a donc pas été mise en œuvre.

5.1.2 Préparation des calculs.

Une fois les maillages obtenus, la chaîne de calcul *SALOME-MPCube* prépare les fichiers nécessaires aux simulations *MPCube*.

Le code de calcul *MPCube*, tout comme le code *Trio-U* sur lequel il est basé, fonctionne grâce à des jeux de données, nommés fichiers *data*. Ces fichiers définissent les caractéristiques des simulations à résoudre : termes sources, conditions aux limites, schéma Volumes Finis utilisé, solveurs, etc. Pour chaque cellule du domaine, un dossier de travail est construit, et un script Python génère ensuite automatiquement le fichier *data* correspondant aux problèmes de cellules et au calcul des matrices locales. Un exemple commenté de jeu de données est disponible à l'annexe §B.2.1.

L'appel à la commande MAKE_PAR.DATA achève la préparation de la simulation. Cette commande native de *Trio-U* construit les fichiers nécessaires à la réso-

lution parallèle du problème. En particulier, elle répartit le domaine de travail, ici le maillage d'une cellule, sur un nombre donné de processeurs et adapte les fichiers *data* en conséquences.

La Figure 5.2 présente un état du dossier de travail consacré à la cellule $(0,0,0)$, une fois terminée la préparation de la simulation.

```
1   ls work/x0/y0/z0:

    CELLULE3D_0_0_0_0000.Zones          Eléments liés au processeur $0$.
    CELLULE3D_0_0_0_0001.Zones          Eléments liés au processeur $1$.
    DEC_data_cellule3D_0_0_0.data       La répartition du milieu sur les processeurs.
6   DEC_data_cellule3D_0_0_0.err
    DEC_data_cellule3D_0_0_0.log
    DEC_data_cellule3D_0_0_0.out
    IntFond.file                        Fichiers décrivant les zones (fond, inclusions)
    IntFond.ssz                         du milieu.
11  IntInclusionD0.file                 Il y a un fichier .file et .ssz par zone.
    IntInclusionD0.ssz
    IntInclusionD1.file
    IntInclusionD1.ssz
    PAR_data_cellule3D_0_0_0.data       Jeu de données pour la résolution parallèle.
16  SEQ_data_cellule3D_0_0_0.data       Jeu de données pour la résolution séquentielle.
    cellule3D_0_0_0.geom                Réecriture du maillage au format geom.
    cellule3D_0_0_0.med                 Maillage original.
    data_cellule3D_0_0_0.data           Jeu de données original.
    ssz.geo                             Paramètres Trio-U.
21  ssz_par.geo                         Paramètres Trio-U.
```

FIGURE 5.2 – Liste des fichiers nécessaires à une simulation *MPCube*. Le domaine *cellule3D_0_0_0*, qui contient trois zones de diffusivités (le fond et deux types d'inclusions), a été préparé pour une résolution parallèle sur 2 processeurs.

La préparation des fichiers de simulation est une tâche indépendante d'une cellule à l'autre. On dispose donc théoriquement d'un parallélisme *extra*-cellulaire. En l'état actuel des choses, celui-ci n'a pas été implémentée, et les cellules sont préparées les unes après les autres.

La commande MAKE_PAR.DATA est séquentielle, son rôle étant justement de préparer un futur travail parallèle. Il n'y a donc pas de parallélisme *intra*-cellulaire possible pour cette étape de la chaîne de calcul.

5.1.3 Résolution des problèmes de cellules.

La résolution des problèmes de cellules s'effectue par la méthode des Volumes Finis *VFDiam*, *via* le code de calcul *MPCube*. En effet, une fois les fichiers *data* correctement préparés, les problèmes de cellules deviennent des problèmes de diffusion classiques. Il n'est donc pas nécessaire d'implémenter de nouvelles fonctionnalités au code *MPCube*.

On dispose donc d'un code de calcul parallèle performant, dont l'implémentation a été qualifiée dans de précédents travaux [61]. D'un point de vue technique, on choisit d'utiliser le solveur BICGSTAB et le préconditionneur multi-grille BOOMERAMG [113] par le biais de la librairie PETSC [154]. Ces solveurs se sont révélés adaptés à la résolution de problèmes de diffusions au sein des matériaux cimentaires [70].

Sur une cellule donnée, on résout les problèmes de cellules les uns après les autres, c'est-à-dire que l'on détermine $\Psi^1_{\hat{K}}$, puis $\Psi^2_{\hat{K}}$, et ainsi de suite. On profite alors du parallélisme *intra*-cellulaire du code de calcul *MPCube* en faisant appel à des solveurs parallèles pour résoudre les problèmes de cellule. En assignant plusieurs processeurs à cette tâche, on s'assure ainsi que chaque problème est résolu efficacement.

Les problèmes de cellules étant indépendants d'une cellule à l'autre, on dispose ici d'un parallélisme *extra*-cellulaire total. Afin de l'exploiter, on fait appel à la fonction *Trio-U* native **Execute_parallel**. Cette fonction permet d'exécuter plusieurs instances de *Trio-U*, et donc de *MPCube*, en parallèle, tout en précisant le nombre de processeurs à allouer à chaque instance.

Ainsi la Figure 5.3 présente un jeu de données permettant de lancer la résolution des problèmes sur trois cellules différentes, chaque cellule se voyant attribuer deux processeurs. Il est possible d'assigner un nombre différent de cœurs de calculs aux différentes instances, ce qui peut être utile si l'une des cellules a une discrétisation plus fine par exemple.

```
Execute_parallel
{
  liste_cas 3 ./Travail/work/x0/y0/z0/PAR_data_cellule3D_0_0_0
              ./Travail/work/x1/y0/z0/PAR_data_cellule3D_1_0_0
                 ./Travail/work/x2/y0/z0/PAR_data_cellule3D_2_0_0
  nb_procs 3 2 2 2
}

FIN
```

FIGURE 5.3 – En exécutant *MPCube* avec ce jeu de données, on lance trois nouvelles instances de *MPCube* qui traiteront les problèmes d'une cellule chacune. On a alloué deux cœurs de calcul à chaque instance.

5.1.4 Construction de la base locale et calcul des matrices locales.

Comme décrit à la section §3.3.1, les solutions $(\Psi^l_{\hat{K}})_{1\leq l\leq n}$ des problèmes de cellules ne forment pas une base de l'espace V_H des solutions du problème grossier (3.2). On construit donc, sur chaque cellule, la base locale de Galerkin $(\Phi^l_K)_{1\leq l\leq n}$ par combinaison linéaire des $(\Psi^l_{\hat{K}})_{1\leq l\leq n}$ (3.25). C'est à partir de ces bases locales que l'on construit une base de Galerkin lors de la résolution du problème grossier.

Une fois les fonctions $(\Phi^l_K)_{1\leq l\leq n}$ obtenues, on calcule les matrices locales de masse, de raideur, de saut, etc. Lors de la résolution du problème grossier, ces matrices sont assemblées, l'assemblage dépendant du schéma choisi pour résoudre le problème grossier : méthode des Éléments Finis (cf. §3.3.1) ou méthode de Galerkin discontinue (cf. §3.3.2).

Le calcul des fonctions $(\Phi^l_K)_{1\leq l\leq n}$ et des matrices locales associées est réalisé par le biais de *MPCube*. La Figure 5.4 présente la liste des instructions *MPCube* utilisées pour effectuer ce travail. Ces commandes doivent être exécutées à la suite de

la résolution des problèmes de cellules, qui sont notés `probleme0` à `probleme7` à la Figure 5.4.

Il ne s'agit pas d'une fonctionnalité native du code de calcul, mais de développements réalisés au cours de ces travaux de thèse. On a développé une série d'objets qui couvre toutes les opérations informatiques nécessaires à l'assemblage du problème grossier. Cela comprend la restriction au macroélément K des fonctions définies sur la cellule $\hat{K}$, l'interpolation des solutions pour obtenir des valeurs sur les faces et sommets du maillage fin $T_h(\hat{K})$, etc. Ces développements sont présentés en détail à la section §5.2.2.

```
Coloration_coordonnee Coloration_macroelement
Lire Coloration_macroelement
{
  expression (x_ge_(0.000000))_and_(x_le_(1.000000))
        _and_(y_ge_(0.000000))_and_(y_le_(1.000000))
        _and_(z_ge_(0.000000))_and_(z_le_(1.000000))
  probleme_associe probleme0
}

Base_locale_DG1HW Phi_K
Lire Phi_K
{
  valeur_rho 0.0
  prefixe cellule3D_0_0_0_
  coloration Coloration_macroelement
  type_sauvegarde ALL
  sommets_macroelement 2 8 3 24
      0.0 0.0 0.0
      1.0 0.0 0.0
      0.0 1.0 0.0
      1.0 1.0 0.0
      0.0 0.0 1.0
      1.0 0.0 1.0
      0.0 1.0 1.0
      1.0 1.0 1.0
  gestionnaire
  {
    dossier_champs /home/simulation/Bases_locales/
    dossier_matrices /home/simulation/Matrices_locales/
    type_sauvegarde ALL
    liste_problemes_locaux
    {
      probleme0 ,
      probleme1 ,
      probleme2 ,
      probleme3 ,
      probleme4 ,
      probleme5 ,
      probleme6 ,
      probleme7
    }
  }
}

Construire_base_locale Phi_K
Calculer_matrices_EF Phi_K
```

FIGURE 5.4 – Commandes *MPCube* permettant la construction de la base locale $(\Phi_K^l)_{1\leq l\leq n}$ et le calcul des matrices locales associées. Ces commandes doivent être exécutées à la suite de la résolution des problèmes de cellules, notés `probleme0`, `probleme1`, etc.

La construction de la base locale et le calcul des matrices locales de *raideur* $\mathbb{K}_K$ (3.38), de *masse* $\mathbb{M}_K$ (3.39) et de *masse de bord* $\mathbb{M}_{b,F}$ (3.41) sont indépendants

d'une cellule à l'autre. On dispose donc pour ces opérations d'un parallélisme *extra*-cellulaire total.

Cependant, si l'on souhaite utiliser une méthode discontinue de Galerkin pour résoudre le problème grossier, on doit également calculer les matrices de *saut* $\mathbb{T}$ (3.47), de *pénalisation* $\mathbb{Y}$ (3.48), de *saut Dirichlet* $\mathbb{T}_b$ (3.49), et de *pénalisation Dirichlet* $\mathbb{Y}_b$ (3.50). Les termes de ces matrices sont les résultats de calculs d'intégrales sur les faces F du maillage grossier $T_H(\Omega)$. Ces calculs font intervenir les valeurs des fonctions de bases sur les deux éléments K^+ et K^- ayant F pour frontières. Les cellules correspondantes $\hat{K}^+$ et $\hat{K}^-$ ne sont donc pas indépendantes.

Pour conserver un parallélisme *extra*-cellulaire complet, on calcule les valeurs des fonctions élémentaires sur les faces de chaque macroélément, puis on stocke ces valeurs. Le calcul proprement dit des matrices de saut et de pénalisation est réalisé séquentiellement lors de l'assemblage du problème grossier.

Pour des raisons techniques, l'objet *MPCube* GESTIONNAIRE_PROJET qui apparaît à la Figure 5.4 nécessite d'avoir en mémoire les objets *MPCube* correspondants aux problèmes de cellules. Pour cette raison, cette étape de la chaîne de calcul doit être réalisée au sein de la même instance *MPCube* que les résolutions des problèmes de cellules. Les choix concernant le parallélisme de cette étape sont donc effectués à l'étape précédente de la chaîne de calcul (cf. §5.1.3). En particulier, la possibilité que plusieurs cœurs de calculs soient affectés à une même cellule (parallélisme *intra*-cellulaire) doit être pris en compte.

Remarque 5.1 *Les besoins de l'objet* GESTIONNAIRE_PROJET *forment une raison supplémentaire de ne pas calculer les termes des matrices de* sauts *et de* pénalisations *à cette étape de la chaîne de calcul. En effet, si on souhaite les calculer maintenant, il est nécessaire de réarranger le travail sur les cellules, non plus macroélément par macroélément, mais face par face. Pour chaque face, la méthode devra alors résoudre les problèmes des deux cellules correspondantes avant de pouvoir calculer les termes matriciels. Au final, chaque problème de cellule aura été résolu quatre fois en dimension deux et six fois en dimension trois. Face à ce gaspillage colossal de ressources, l'idée a donc été abandonnée.*

5.1.5 Mise en forme des résultats locaux.

La résolution des problèmes de cellules à l'étape précédente de la chaîne de calcul *SALOME-MPCube* conduit à la création de nombreux fichiers. Outre les fichiers temporaires créés par *MPCube*, on trouve les enregistrements de la base locale $(\Phi_K^l)_{1\leq l\leq n}$, ainsi que les matrices locales correspondantes. Il est également possible de stocker le flux de la base locale $(D\cdot\nabla\Phi_K^l)_{1\leq l\leq n}$.

La base locale $(\Phi_K^l)_{1\leq l\leq n}$ est sauvegardée au format LATA. Il s'agit du format le plus simple à manipuler au sein du code de calcul *MPCube*, ce qui en a fait le format privilégié des outils *MPCube* développés durant ces travaux de thèse. Le format LATA sauvegarde cependant chaque fonction Φ_K^l dans un fichier indépendant, ce

qui entraîne la création d'un grand nombre de fichiers lors de la résolution des problèmes locaux.

Afin d'économiser les ressources informatiques, on convertit les fichiers LATA au format MED. Ce format permet de conserver dans un seul fichier la discrétisation de la cellule $T_h(\hat{K})$ et les valeurs des fonctions $(\Phi_K^l)_{1\leq l\leq n}$ sur ce maillage. Cette conversion se déroule en deux étapes.

Tout d'abord, il y a la conversion proprement dite, c'est-à-dire la création d'un fichier MED à partir des fichiers LATA. Cette conversion est le fait d'une fonction native de *MPCube*, la méthode **Lata_To_Med**. Il suffit donc de lancer une instance de *MPCube* avec le jeu de données adéquat pour obtenir le résultat voulu. La Figure 5.5 présente un exemple d'un tel jeu de données.

Le fichier MED ainsi créé contient, outre les valeurs des fonctions $(\Phi_K^l)_{1\leq l\leq n}$, la discrétisation de la cellule $T_h(\hat{K})$. Cependant, la numérotation de cette discrétisation, sommets et volumes, n'est pas celle du maillage d'origine fournit par *SALOME*, mais une numérotation automatiquement construite par **Lata_To_Med**. On renumérote donc le maillage du fichier MED pour le faire correspondre à la numérotation originale. Cette renumérotation s'effectue par le biais de *MPCube*, le jeu de données correspondant étant reproduit à l'Annexe §B.2.2.

```
dimension 3

Lata_To_Med  /home/simulation/wX0/wY0/wZ0/cellule3D_0_0_0_Solution.lata
             /home/simulation/wX0/wY0/wZ0/cellule3D_0_0_0_Solution.med

FIN
```

FIGURE 5.5 – Commandes *MPCube* permettant la conversion d'un ensemble de fichiers LATA en un fichier MED.

La fonction **Lata_To_Med** ne fonctionne que séquentiellement : il n'est pas possible d'assigner plusieurs cœurs de calcul à cette tâche. De même, la renumérotation ne peut s'effectuer que séquentiellement, puisque le cœur de calcul a besoin de la discrétisation complète du milieu pour travailler, et non pas seulement d'une portion. Les problèmes *MPCube* de conversion et de renumérotation doivent donc tous deux être réalisés séquentiellement. La mise en forme des résultats ne dispose donc pas de parallélisme *intra*-cellulaire.

Cependant, cette mise en forme est indépendante d'une cellule à l'autre. On dispose donc d'un parallélisme *extra*-cellulaire complet, que l'on peut exploiter grâce à la fonction *MPCube* **Execute_parallel**. La Figure 5.6 présente un jeu de données permettant de mettre en forme simultanément les résultats de deux cellules. Ce jeu de donnée est très proche de celui permettant de résoudre les problèmes sur plusieurs cellules simultanément, présentés à la Figure 5.3. Contrairement à ce dernier cependant, on ne peut allouer ici qu'un seul cœur de calcul à chaque cellule.

```
dimension 3

Execute_parallel
{
  liste_cas 2 /home/simulation/wX0/wY0/wZ0/cellule3D_0_0_0_conversion
              /home/simulation/wX1/wY0/wZ0/cellule3D_1_0_0_conversion
  nb_procs 2 1 1
}

Execute_parallel
{
  liste_cas 2 /home/simulation/wX0/wY0/wZ0/cellule3D_0_0_0_projection
              /home/simulation/wX1/wY0/wZ0/cellule3D_1_0_0_projection
  nb_procs 2 1 1
}

FIN
```

FIGURE 5.6 – En exécutant *MPCube* avec ce jeu de données, on met en forme les résultats des cellules $(0,0,0)$ et $(1,0,0)$. On ne peut allouer ici qu'un seul cœur de calcul à chaque instance.

Remarque 5.2 *À ce stade de la chaîne de calcul* SALOME-MPCube*, tous les calculs préliminaires à la résolution du problème grossier sont terminés. On dispose non seulement d'une base de Galerkin, mais également des matrices locales, qui permettront d'assembler, puis de résoudre, le problème grossier (3.2). Ces calculs dépendent entre autres de la diffusivité D et du découpage choisi mais sont indépendants du problème grossier à résoudre. À domaine de travail* Ω *et maillage grossier* $T_H(\Omega)$ *fixés, on peut donc résoudre n'importe quel problème grossier.*

5.1.6 Assemblage et résolution du problème grossier.

Au sein de la chaîne de calcul *SALOME-MPCube*, deux méthodes sont disponibles pour résoudre le problème grossier (3.2) : la méthode des Éléments Finis, présentée à la section §3.3.1, et la méthode de Galerkin discontinue, décrite à la section §3.3.2. Une fois le problème matriciel assemblé et résolu, on dispose de C_H, la solution grossière du problème.

Aucune des méthodes grossières, Éléments Finis ou Galerkin discontinue, n'est disponible nativement dans le code *MPCube*, celui-ci se concentrant sur les méthodes Volumes Finis. Au cours de ces travaux de thèse, on a donc implémenté ces méthodes dans *MPCube*. Cette extension n'est pas indépendante mais conçue, au contraire, pour s'interfacer aux méthodes multi-échelles. Ainsi l'assemblage du problème matriciel fait appel aux matrices locales précédemment calculées. La résolution proprement dite du problème matriciel fait cependant appel aux solveurs disponibles dans *MPCube*.

La Figure 5.7 présente un jeu de donnés permettant de résoudre un problème grossier. L'implémentation des méthodes grossières et les exemples permettant de qualification sont détaillés à la section §5.2.3.

Par sa nature même, l'assemblage et la résolution du problème grossier ne demande aucun travail local. Les notions de parallélismes *intra*-cellulaire et *extra*-

```
dimension 3
Probleme_grossier_EFP1 mon_probleme_grossier
Lire mon_probleme_grossier
   {
    decoupage 3 5 5 5
    extremites 2 2 3 6 0.0 0.0 0.0 5.0 5.0 5.0
    dossier_travail /home/simulation/Travail/
    dossier_matrices_locales /home/simulation/Matrices_locales/
    terme_source 0
    conditions_limites
      {
         zm Dirichlet 1
         zp Dirichlet 0
         ym Neumann 0
         yp Neumann 0
         xm Neumann 0
         xp Neumann 0
      }
    solveur gen
       {
        seuil 1e-13
        impr
        solv_elem bicgstab precond ilu
            {
             type    2
             filling 10
            }
       }
   }
Resoudre_probleme_grossier mon_probleme_grossier
FIN
```

FIGURE 5.7 – Jeu de données permettant de résoudre un problème grossier. Les paramètres `decoupage` et `dossier_matrices_locales` permettent de lier le problème grossier au reste de la chaîne multi-échelle *SALOME-MPCube*.

cellulaire n'ont donc pas de ce sens à ce stade la chaîne de calcul.

Il est cependant possible d'assigner plusieurs cœurs de calculs à la résolution du problème grossier, et de profiter ainsi des capacités des solveurs parallèles de *MPCube*. Ce parallélisme s'apparente au parallélisme *intra*-cellulaire, au sens où plusieurs cœurs de calculs sont utilisés pour résoudre une tâche. Le parallélisme *extra*-cellulaire, au contraire, permet de résoudre de front plusieurs tâches distinctes.

En pratique cependant les problèmes grossiers sont de tailles réduites : ils ne dépassent que rarement la centaine de milliers d'inconnues. L'intérêt d'une résolution parallèle du problème grossier est donc limité et n'a pas été poursuivi.

5.1.7 Reconstruction de la solution fine.

La dernière étape de la chaîne *SALOME-MPCube* calcule au niveau fin la solution $C_{H,h}$, en pondérant les fonctions $(\Phi^I)_{I\in\mathscr{I}}$ par les valeurs de la solution grossière C_H (3.52).

Cette reconstruction n'intervient généralement que sur une portion restreinte du domaine. En effet, on résout ici des problèmes de *très grandes tailles* pour lesquels une résolution classique est techniquement irréalisable. Il est donc généralement impossible de sauvegarder, et à plus fortes raisons de visualiser, la solution fine $C_{H,h}$ d'un seul tenant, tant la somme des discrétisations fines $(T_h(K))_{K\in T_H(\Omega)}$ est immense. Il peut cependant être intéressant de reconstruire la solution sur une zone déterminée, par exemple autour d'une singularité. On ne manipule ainsi que des

blocs de tailles raisonnables, qui n'excèdent pas les capacités informatiques disponibles.

Sur le plan technique, la reconstruction de la solution fine se déroule en trois étapes. Dans un premier temps, le solution est calculée sur chaque macroélément K à partir des valeurs des fonctions $(\Phi_K^l)_{1\leq l\leq n}$, sauvegardées au format MED. On utilise pour cela les librairies SMESH et MEDMEM de la plate-forme *SALOME* qui permettent de manipuler les données contenus dans un fichier MED : visualisation de la discrétisation, opérations arithmétiques sur les champs de données solutions, etc.

En théorie, les fonctions $(\Phi_K^l)_{1\leq l\leq n}$ sont définies sur le macroélément K. En pratique cependant, elles sont stockées sur la même discrétisation que les fonctions $(\Psi_{\hat{K}}^l)_{1\leq l\leq n}$ qui ont servi à les calculer, c'est-à-dire sur la discrétisation fine $T_h(\hat{K})$ de la cellule. Il est donc nécessaire de restreindre les fonctions $(\Phi_K^l)_{1\leq l\leq n}$ à la seule discrétisation $T_h(K)$ du macroélément sous-jacent. On fait appel pour cela à MEDSPLITTER, un outil de la plate-forme *SALOME*. MEDSPLITTER permet, comme son nom l'indique, de diviser un maillage, et les champs associés, en plusieurs blocs.

Enfin on fusionne les solutions fines, maintenant disponibles macroélément par macroélément, en une seule discrétisation. Pour cela, on fait à nouveau appel à MEDSPLITTER. Il est en effet possible, avec le paramétrage adéquat, d'utiliser cette commande pour agréger des maillages. Schématiquement, les maillages, ainsi que les champs associés, sont placés côte à côte, puis renumérotés par blocs. MEDSPLITTER ne se soucie pas de relier les maillages entre eux, ni de fusionner des sommets ou des faces qui seraient confondues. Au terme de cette opération, on dispose cependant d'un unique fichier MED contenant le champ solution sur un patch de macroéléments.

5.2 Implémentation et qualification.

On détaille dans cette section l'implémentation de la chaîne de calcul multi-échelle au sein du code de calcul *MPCube*, en commençant par présenter *MPCube* et le noyau *Trio-U* (§5.2.1).

Compte tenu des outils disponibles dans la version native de *MPCube*, il n'est pas nécessaire de développer un outil spécifique pour résoudre les problèmes de cellules et on peut obtenir les fonctions $(\Psi_{\hat{K}}^l)_{1\leq l\leq n}$ par le biais des jeux de données appropriés.

La construction des bases locales $(\Phi_K^l)_{1\leq l\leq n}$ et le calcul des matrices locales associées demandent par contre un important travail d'implémentation, que l'on décrit à la section §5.2.2.

Enfin, on présente à la section §5.2.3 la structure encadrant la résolution des problèmes grossiers par les méthodes d'Éléments Finis et de Galerkin discontinue. Chaque partie comprend les résultats d'un ou plusieurs exemples de simulations qui permettent de qualifier les implémentations.

5.2.1 Le code de calcul multi-physique *MPCube*.

À l'origine du code de calcul *MPCube* se trouve le noyau *Trio-U* [171]. *Trio-U* est un logiciel de simulation numérique en mécanique des fluides (code CFD pour *Computational Fluid Dynamics*). Il est développé au Laboratoire de Modélisation et de Développement Logiciels (LMDL) de la Direction de l'Énergie Nucléaire (DEN) du Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA).

Ce noyau est implémenté en C++ [80] selon une approche objet. Il propose un cadre de travail générique pour le développement de codes de calculs, notamment par la définition de notions élémentaires communes : problème, équation, opérateur, schéma en temps, discrétisation, etc. Le noyau *Trio-U* dispose d'une interface vers différentes bibliothèques de solveurs linéaires, telles que HYPRE [123] ou PETSC [154].

Trio-U fournit également les structures de données et les fonctionnalités nécessaires à la conception de codes de calculs parallèles. Il permet ainsi une gestion de la mémoire et des entrées sorties parallèles, des opérations distribuées sur les vecteurs et les matrices et des algorithmes de résolution distribuée des systèmes linéaires. Ces outils parallèles sont internes au code et prêts à l'emploi.

Outre le code de calcul *MPCube*, le noyau *Trio-U* est utilisé par un autre code du CEA/DEN : le code *Flica-Ovap* [130].

Le code de calcul multi-physique *MPCube* est développé au Laboratoire de Simulation des Écoulements et Transports (LSET) du CEA Saclay. Il a été initialement développé pour la simulation des écoulements multi-phasiques en milieux poreux mais de manière plus générale est adapté aux problèmes de diffusion-convection dans les milieux poreux : hydraulique, transport d'espèces, écoulements multi-phasiques, etc. Une importante phase de qualification a été récemment réalisée, notamment une vérification de l'ordre des schémas et une analyse de la scalabilité [19, 60, 62].

MPCube implémente notamment la méthode de Volumes Finis *VFDiam* [20]. Cette méthode, qui est détaillée à la section §3.2.3, a la particularité d'être utilisable en dimension deux et trois, et ce quelle que soit la forme des éléments de la discrétisation : triangle ou quadrangle en $2D$, tétraèdre ou héxahèdre en $3D$. Ce schéma est bien adapté au traitement des discontinuités des paramètres physiques. On l'utilise dans ces travaux de thèse pour résoudre les problèmes de diffusion au sein des matériaux cimentaires.

5.2.2 Construction des fonctions locales de Galerkin.

On présente maintenant les outils mis en place au sein du code de calcul *MPCube* afin de construire, sur chaque macroélément K du maillage grossier $T_H(\Omega)$, la base locale $(\Phi_K^l)_{1\leq l\leq n}$ à partir des solutions des problèmes de cellules $(\Psi_{\hat{K}}^l)_{1\leq l\leq n}$, puis de calculer les matrices locales associées.

Il s'agit de l'implémentation de l'étape §5.2.2 de la chaîne multi-échelle *SALOME-MPCube*. Elle a été réalisé en C++, et utilise le cadre de travail générique de *Trio-U*.

5.2.2.1 Présentation de l'objet COLORATION.

Le taux de sur-échantillonnage ρ est un paramètre important des méthodes multi-échelles développées au cour de ces travaux de thèse. Quand ρ est strictement positif, la chaîne de calcul doit manipuler deux domaines distincts : la cellule $\hat{K}$, sur laquelle sont définies les solutions des problèmes de cellules $(\Psi^l_{\hat{K}})_{1\leq l\leq n}$, et le macroélément K, sur lequel est définie la base locale $(\Phi^l_K)_{1\leq l\leq n}$. Par définition, le macroélément K est inclus dans la cellule $\hat{K}$.

On dispose d'une discrétisation fine $T_h(\hat{K})$ de la cellule $\hat{K}$, qui a permis la résolution numériques des problèmes de cellules. Afin de disposer, au sein de l'instance *MPCube*, d'une discrétisation $T_h(K)$ du macroélément K, on doit identifier les éléments k de $T_h(\hat{K})$ qui appartiennent à K. C'est le rôle de l'objet COLORATION.

L'objet COLORATION permet de *peindre* les volumes d'une discrétisation selon un critère donnée, dans le cas présent l'appartenance au macroélément K. Il est possible de peindre des volumes en fonction de leur groupe MED (cf. §6.2.2.4) mais aussi des coordonnées de leurs sommets ou de leur barycentre.

L'objet COLORATION permet également de construire la frontière de la zone peinte. Pour une face f de $T_h(\hat{K})$ si un, et un seul, des volumes auxquels appartient f est coloré, alors f appartient à la frontière de la zone peinte. Cette fonctionnalité permet d'identifier la discrétisation $T_h(\partial K)$ de la frontière ∂K du macroélément K.

Remarque 5.3 *Afin de pouvoir colorer le macroélément K dans la discrétisation $T_h(\hat{K})$ de la cellule $\hat{K}$, il est nécessaire que cette discrétisation soit* adaptée *au macroélément. On rappelle qu'un maillage est dit* adapté *si tout élément du maillage ; volume, face ou sommet ; appartient à un unique objet physique : le macroélément, la zone de recouvrement, une interface du domaine, etc. Cette contrainte est prise en compte lors de la génération des maillages de cellules, voir à ce propos le chapitre §6 et notamment la section §6.1.1.*

5.2.2.2 Présentation de l'objet GESTIONNAIRE_PROJET.

Lors d'une simulation *MPCube*, chaque problème de cellule est contenu dans un objet MPCUBE_PROBLEME_BASE indépendant des autres. On a développé l'objet GESTIONNAIRE_PROJET afin de pouvoir manipuler ces problèmes d'un seul tenant. Par le biais cet objet, on peut accéder aux valeurs des fonctions $(\Psi^l_{\hat{K}})_{1\leq l\leq n}$, ainsi qu'aux propriétés de la discrétisation de la cellule, comme par exemple les volumes des éléments.

De manière générale, le rôle de l'objet GESTIONNAIRE_PROJET est de faire le lien entre les problèmes de cellule et l'objet BASE_LOCALE qui, lui, construit

la base locale $(\Phi_K^l)_{1\leq l\leq n}$. Dans cette optique, GESTIONNAIRE_PROJET permet notamment d'interpoler les valeurs des fonctions $(\Psi_{\hat{K}}^l)_{1\leq l\leq n}$ aux faces et aux sommets du maillage, *via* les équations (3.23) et (3.24). Il permet également d'accéder aux valeurs des flux $(D\cdot\nabla\Psi_{\hat{K}}^l)_{1\leq l\leq n}$ calculées par le schéma Volumes Finis *VFDiam* selon l'approximation (3.22).

5.2.2.3 Calcul des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ par l'objet BASE_LOCALE.

L'objet BASE_LOCALE permet de calculer les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ sur chaque macroélément $K\in T_H(\Omega)$. Du point de vue informatique, il possède un objet GESTIONNAIRE_PROJET, qui lui donne accès aux problèmes de cellule et leurs solutions, et un objet *MPCube*Coloration, qui lui permet de distinguer le macroélément K au sein de la discrétisation $T_h(\hat{K})$ de $\hat{K}$.

On peut trouver à la Figure 5.4 le jeu de données *MPCube* permettant d'effectuer cette opération, tandis que l'Algorithme 5.1 résume les différentes étapes de travail.

Algorithme 5.1 Objet BASE_LOCALE : construction de la base locale $(\Phi_K^l)_{1\leq l\leq n}$.

1: Déterminer les sommets $(S_i)_{1\leq i\leq n}$ de K.
2: Interpoler les fonctions $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$ en les points $(S_i)_{1\leq i\leq n}$.
3: Résoudre le problème linéaire (5.1).
4: Calculer la base locale $(\Phi_K^i)_{1\leq i\leq n}$.
5: Sauvegarder la base locale au format LATA.

La base locale $(\Phi_K^i)_{1\leq i\leq n}$ est identique que l'on utilise une méthode d'Éléments Finis ou de Galerkin discontinue à l'échelle grossière. Les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ sont combinaisons linéaires des fonctions $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$, les solutions des problèmes de cellules (3.6). On détermine les coefficients $a_{i,l}$ en imposant, pour tout $1\leq i\leq n$:

$$\forall 1\leq j\leq n \qquad \Phi_K^i(S_j)=\sum_{1\leq l\leq n} a_{i,l}\Psi_{\hat{K}}^l(S_j)=\delta_{i,j}, \tag{5.1}$$

où on a repris les notations de la section §3.2.3, i.e on a noté $(S_i)_{1\leq i\leq n}$ les sommets du macroélément K et δ le symbole de Kronecker. On fournit à *MPCube* les coordonnées des points $(S_i)_{1\leq i\leq n}$ par le biais du jeu de données.

Avant de résoudre le problème linéaire (5.1), il est nécessaire d'obtenir les valeurs des fonctions $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$ en chacun des sommets $(S_i)_{1\leq i\leq n}$. La méthode *VFDiam* étant une méthode Volumes Finis centrée, la résolution numérique des problèmes de cellule permet d'obtenir une valeur sur chaque volume k de $T_h(\hat{K})$, et non sur les sommets. Lors de l'assemblage du schéma, elle approxime cependant les valeurs aux sommets, utilisant pour cela une méthode des moindres carrés (3.23).

Sur le plan informatique, les problèmes de moindres carrés ont été résolus pendant la résolution des problèmes de cellules, puis conservés en mémoire au sein de l'objet MPCUBE_PROBLEME_BASE correspondant. Les coefficients permettant de calculer $\Psi^i_{\hat{K}}(S_j)$ en fonction des valeurs de $\Psi^i_{\hat{K}}$ sur les éléments voisins de S_j sont donc disponibles sans besoin d'un travail supplémentaire. On accède donc à ces coefficients par le biais de l'objet GESTIONNAIRE_PROJET, ce qui permet d'assembler le problème linéaire (5.1).

Le problème (5.1) assemblé et résolu, on calcule la base locale $(\Phi^i_K)_{1\leq i\leq n}$. Ces fonctions, bien que définies en théorie sur le macroélément K, sont calculées sur la cellule $\hat{K}$. En effet, au sein de *MPCube*, les manipulations des champs solutions (multiplication par un scalaire, addition) sont aisées si les champs sont définis sur un même domaine. Le passage de la cellule au macroélément s'effectue en pratique lors de l'étape de mise en forme des résultats, que l'on a décrite à la section §5.1.5. Les fonctions $(\Phi^i_K)_{1\leq i\leq n}$ sont ensuite sauvegardées au format LATA. Elles sont ensuite mises en formes au format MED pour être utilisées plus loin dans la chaîne de calcul *SALOME-MPCube*, lors du recalcul de la solution fine $C_{H,h}$ (cf. §5.1.7).

Remarque 5.4 *Contrairement au format* MED*, le format* LATA *stocke les informations à travers plusieurs fichiers. La sauvegarde de la base locale conduit ainsi à la création de $n+1$ fichiers : un pour le maillage $T_h(\hat{K})$ et un par champ solution Φ^i_K. Le format* LATA *étant de plus incompatible avec la plate-forme* SALOME*, il n'apparaît pas comme un choix judicieux de format de stockage.*

Il s'agit en fait d'un choix d'implémentation. La sauvegarde au format LATA *est en effet très facile à implémenter au sein d'un objet* MPCube*. Au contraire, le format* MED *qui aurait été ici le choix évident, est étroitement lié aux mécaniques et objets fondamentaux de* Trio-U*. Il est donc difficile de le manipuler hors du cadre précis pour lequel il a été implémenté à l'origine.*

5.2.2.4 Calcul des matrices locales par l'objet BASE_LOCALE.

Afin de résoudre le problème grossier par une méthode d'Éléments Finis, on doit calculer les différents termes des matrices locales de raideur $\mathbb{K}_K$ (3.38), de masse $\mathbb{M}_K$ (3.39) et de masse de bord $\mathbb{M}_{b,F}$ (3.41) pour un élément $K \in T_H(\Omega)$ et une face $F \in T_H(\partial\Omega)$ donnés. Ces quantités sont définies par :

$$\forall 1 \leq i,j \leq n \qquad \mathbb{K}_K(i,j) = \frac{1}{2} \sum_{f \in T_h(\partial K)} \left(\Phi_K^i(f) \int_f D\nabla\Phi_K^j \cdot \mathbf{n} + \Phi_K^j(f) \int_f D\nabla\Phi_K^i \cdot \mathbf{n} \right) \tag{5.2}$$

$$\forall 1 \leq i,j \leq n \qquad \mathbb{M}_K(i,j) = \sum_{k \in T_h(K)} |k| \Phi_K^i(k) \Phi_K^j(k) \tag{5.3}$$

$$\forall 1 \leq i,j \leq n \qquad \mathbb{M}_{b,F}(i,j) = \sum_{f \in T_h(F)} |f| \Phi_K^i(f) \Phi_K^j(f) \tag{5.4}$$

Une description précise de ces équations et un rappel des différentes notations est disponible à la section §3.3.1.

Algorithme 5.2 Objet BASE_LOCALE : calcul des matrices locales.

1: Coloration du macroélément K et de sa frontière.
2: Interpolation des fonctions $(\Phi_K^i)_{1 \leq i \leq n}$ aux faces de $T_h(\partial K)$.
3: Calculer le flux des fonctions $(\Phi_K^i)_{1 \leq i \leq n}$ aux faces de $T_h(\partial K)$.
4: Calculer les matrices locales.
5: Sauvegarder les matrices locales.
6: Sauvegarder les valeurs interpolées et le flux aux faces de $T_h(\partial K)$.

Ces matrices sont calculées par l'objet BASE_LOCALE en suivant les étapes de l'Algorithme 5.2. On commence par identifier les éléments k de $T_h(\hat{K})$ constituant une discrétisation $T_h(K)$ de K, ainsi que les faces f de $T_h(\hat{K})$ qui discrétisent $T_h(\partial K)$. Cette tâche est dévolue à l'objet COLORATION que l'on a décrit à la section §5.2.2.1.

Pour calculer les matrices locales de raideur $\mathbb{K}_K$ et de masse de bord $\mathbb{M}_{b,F}$, il est nécessaire d'obtenir les valeurs des fonctions $(\Phi_K^i)_{1 \leq i \leq n}$ en chacune des faces de $T_h(\partial K)$. La méthode *VFDiam* étant une méthode Volumes Finis centrée, la résolution numérique des problèmes de cellule permet d'obtenir une valeur sur chaque volume k de $T_h(\hat{K})$, et non sur les faces. Au cours de la construction théorique du schéma *VFDiam*, une valeur au barycentre G_f de la face f est calculée afin d'imposer l'égalité des flux à travers la face (3.21).

On utilise l'équation (3.21) pour interpoler les valeurs des fonctions $(\Phi_K^i)_{1 \leq i \leq n}$ aux faces $f \in T_h(\partial K)$. Les valeurs aux faces sont des combinaisons linéaires de valeurs aux éléments, que l'on connaît, et des valeurs aux sommets, dont on a expliqué l'interpolation à la section précédente §5.2.2.3.

Les coefficients de ces combinaisons linéaires font intervenir la diffusivité D du milieu et les propriétés géométriques (surface, orientation) des faces. Sur le

plan informatique, ils ont été calculés pendant la résolution des problèmes de cellules, puis conservés en mémoire au sein des objets MPCUBE_PROBLEME_BASE correspondants. On obtient donc ces coefficients par le biais de l'objet GESTIONNAIRE_PROJET sans besoin de travail supplémentaire.

La seconde quantité nécessaire au calcul de la matrice locale de raideur $\mathbb{K}_K$ est le flux des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ à travers les faces f de $T_h(\partial K)$. Le flux des fonctions $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$ est calculé par le schéma *VFDiam* lors de la résolution des problèmes de cellules (cf. (3.22)). Par le biais de l'objet GESTIONNAIRE_PROJET, on récupère les valeurs de ces flux, puis on calcule le flux des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ par sommation, les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ étant combinaison linéaire des fonctions $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$.

Une fois que l'on dispose de toutes les quantités nécessaires, on calcule les matrices locales. On les sauvegarde ensuite afin de pouvoir les utiliser lors de la résolution du problème grossier.

Si on souhaite utiliser une méthode de Galerkin discontinue pour résoudre le problème grossier, on doit disposer des matrices de *saut* $\mathbb{T}_K$ (3.47), de *pénalisation* $\mathbb{Y}_K$ (3.48), de *saut Dirichlet* $\mathbb{T}_{b,F}$ (3.49), et de *pénalisation Dirichlet* $\mathbb{Y}_{b,F}$ (3.50). Une description précise de ces matrices est disponible à la section §3.3.2.3.

Comme on l'a expliqué à la section §5.1.4, et plus précisément à la Remarque 5.1, ces matrices ne sont pas calculées ici, mais lors de l'assemblage du problème grossier. En effet, la définition de ces matrices s'appuie sur le *saut* et la *moyenne* des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ à travers les faces F de $T_H(\Omega)$. Le saut et la moyenne sur une face F, définis par la Définition 3.5, font appel aux valeurs issues des deux côtés de F, en l'occurence les valeurs des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ sur deux macroéléments voisins.

Afin de pouvoir paralléliser la résolution des problèmes de cellules, chaque instance *MPCube* ne contient les informations relatives qu'à un unique macroélément. Il n'est donc pas possible d'accéder aux fonctions de bases d'un macroélément voisin. On ne peut par conséquent pas calculer les matrices locales $\mathbb{T}_K$, $\mathbb{Y}_K$, $\mathbb{T}_{b,F}$, $\mathbb{Y}_{b,F}$ à ce stade de la chaîne de calcul.

On sauvegarde donc les valeurs interpolées aux faces et le flux des fonctions $(\Phi_K^i)_{1\leq i\leq n}$ pour pouvoir calculer ces matrices lors de l'assemblage du problème grossier.

5.2.2.5 Qualification de l'objet BASE_LOCALE.

On valide dans cette section l'implémentation de l'objet BASE_LOCALE et des objets COLORATION et GESTIONNAIRE_PROJET.

Le calcul de la base locale $(\Phi_K^i)_{1\leq i\leq n}$ à partir des solutions des problèmes de cellules $(\Psi_{\hat{K}}^i)_{1\leq i\leq n}$ est simple à valider. Il suffit en effet de vérifier que les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ valent bien 1 ou 0 sur les sommets $(S_l)_{1\leq l\leq n}$.

On se concentre donc sur le calcul des termes matriciels qui, outre les fonctions de bases $(\Phi_K^i)_{1\leq i\leq n}$, fait intervenir les formules d'interpolations aux sommets et aux faces des éléments et la coloration des éléments du maillage $T_h(\hat{K})$.

Exemple 5.1 *On considère dans un premier temps, le cas linéaire définit par :*

$$D = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$
$$K =]0,1[\times]0,1[\times]0,1[$$
$$F = \{x = 0\}$$

où a, b et c sont des réels strictements positifs. Dans ce cas, les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ sont explicitement connues, et ce pour toute valeur de ρ :

$$\Phi_K^1 = (1-x)(1-y)(1-z) \qquad \Phi_K^5 = (1-x)(1-y)z$$
$$\Phi_K^2 = x(1-y)(1-z) \qquad \Phi_K^6 = x(1-y)z$$
$$\Phi_K^3 = (1-x)y(1-z) \qquad \Phi_K^7 = (1-x)yz$$
$$\Phi_K^4 = xy(1-z) \qquad \Phi_K^8 = xyz$$

Les matrices $\mathbb{K}_K$, $\mathbb{M}_K$ et $\mathbb{M}_{b,F}$ sont symétriques. De plus, étant donnée la symétrie des fonctions $(\Phi_K^i)_{1\leq i\leq n}$, de nombreux termes de ces matrices sont identiques. On a ainsi :

$$\mathbb{M}_K(1,2) = \mathbb{M}_K(1,3) = \mathbb{M}_K(1,5),$$
$$\mathbb{M}_K(1,4) = \mathbb{M}_K(1,6) = \mathbb{M}_K(1,7),$$

des formules semblables existant pour les matrices $\mathbb{K}_K$ et $\mathbb{M}_{b,F}$. Les Figures 5.8, 5.9 et 5.10 présentent le maximum de l'erreur entre les valeurs théoriques ($\mathbb{K}_K$, $\mathbb{M}_K$ et $\mathbb{M}_{b,F}$) et numériques ($\tilde{\mathbb{K}_K}$, $\tilde{\mathbb{M}_K}$ et $\tilde{\mathbb{M}_{b,F}}$) des matrices locales :

$$e_\infty(\mathbb{A}) = \sup_{1\leq i,j\leq n} |\tilde{\mathbb{A}}(i,j) - \mathbb{A}(i,j)| \tag{5.5}$$

Il apparaît des simulations que les erreurs décroissent en $\mathcal{O}(h^2)$.

Exemple 5.2 *Le second exemple que l'on considère est le suivant :*

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$
$$K =]0,1[^3$$
$$F = \{x = 0\}$$
$$\beta = \cos(\pi x)\cos(\pi y)\cosh(\pi z)$$

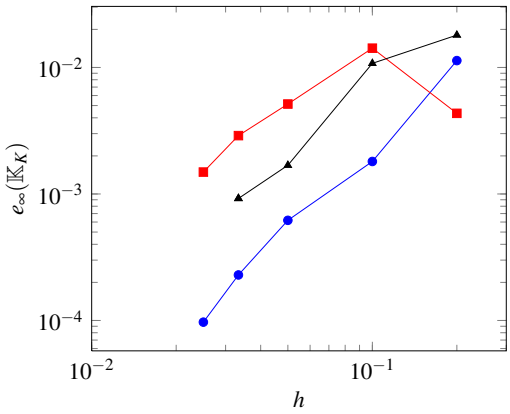

FIGURE 5.8 – Exemple 5.1 : maximum de l'erreur de calcul, en fonction de h, des coefficients de $\mathbb{K}_K$ pour $\rho = 0$ (—●—), $\rho = 0.05$ (—■—) et $\rho = 0.1$ (—▲—).

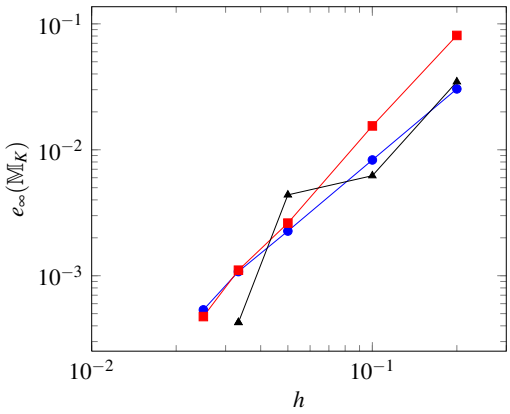

FIGURE 5.9 – Exemple 5.1 : maximum de l'erreur de calcul, en fonction de h, des coefficients de $\mathbb{M}_K$ pour $\rho = 0$ (—●—), $\rho = 0.05$ (—■—) et $\rho = 0.1$ (—▲—).

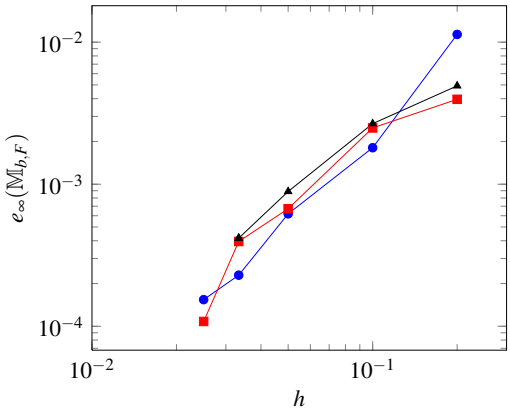

FIGURE 5.10 – Exemple 5.1 : maximum de l'erreur de calcul, en fonction de h, des coefficients de $\mathbb{M}_{b,F}$ pour $\rho = 0$ (—●—), $\rho = 0.05$ (—■—) et $\rho = 0.1$ (—▲—).

La solution du problème est alors :

$$\Psi = \cos(\pi x)\cos(\pi y)\cosh(\pi z)$$

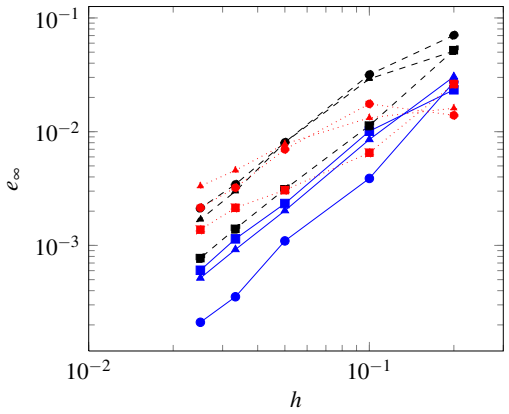

FIGURE 5.11 – Exemple 5.2 : erreur de calcul, en fonction de h, des coefficients $\mathbb{K}_K(\Psi,\Psi)$ (—●—), $\mathbb{M}_K(\Psi,\Psi)$ (—■—) et $\mathbb{M}_{b,F}(\Psi,\Psi)$ (—▲—) pour $\rho = 0$ (trait), $\rho = 0.5$ (pointillé) et $\rho = 0.1$ (tiret).

On peut alors calculer les termes (5.2)-(5.4) en posant $\Phi_K^i = \Phi_K^j = \Psi$. On présente à la Figure 5.11 l'erreur entre valeurs théoriques et numériques de ces trois termes pour différentes valeurs de ρ. Pour $\rho = 0$, les coefficients $\mathbb{K}(\Psi,\Psi)$ et $\mathbb{M}(\Psi,\Psi)$ et $\mathbb{M}_b(\Psi,\Psi)$ convergent en $\mathscr{O}(h^2)$. Quand le sur-échantillonnage est non nul, la convergence varie entre $\mathscr{O}(h^1)$ et $\mathscr{O}(h^2)$ selon les termes.

5.2.3 Résolution du problème grossier.

5.2.3.1 L'objet PROBLEME_GROSSIER.

Le code *MPCube* natif n'implémente aucune méthode d'Éléments Finis ou de Galerkin discontinue, y préférant les méthodes de Volumes Finis comme le schéma *VFDiam* par exemple. Même dans le cas contraire, il aurait été fort peu probable que ces méthodes soient compatibles avec la chaîne de calcul multi-échelle *SALOME-MPCube*. En effet, le principe même des méthodes multi-échelles est d'utiliser, pour résoudre le problème grossier, de fonctions issues des problèmes de cellules, et non d'une base usuelle comme les polynômes de Legendre.

Afin de pallier à ce manque, l'objet PROBLEME_GROSSIER a donc été ajouté au code *MPCube*. Il est déclinable en deux versions, respectivement les objets PROBLEME_GROSSIER_EFP1 et PROBLEME_GROSSIER_GAP1, qui permettent de résoudre un problème grossier dans un contexte multi-échelle par les méthodes d'Éléments Finis (cf. §3.3.1) et de Galerkin discontinue à pénalité intérieure (cf. §3.3.2) respectivement.

On trouve à la Figure 5.7 le jeu de données commandant la résolution d'un problème grossier par une méthode Éléments Finis. L'Algorithme 5.3 récapitule les étapes de cette résolution.

Algorithme 5.3 Objet PROBLEME_GROSSIER : résolution du problème grossier.

1: Assembler le problème grossier à partir des matrices locales.
2: Résoudre le problème grossier *via* un solveur de *MPCube*.
3: Sauvegarder la solution C_H.

Afin de résoudre le problème grossier, on assemble le problème matriciel correspondant à partir des matrices locales. Les matrices locales de raideur $\mathbb{K}_K$ (3.38), de masse $\mathbb{M}_K$ (3.39) et de masse de bord $\mathbb{M}_{b,F}$ (3.41) ont été calculées lors de la résolution des problèmes de cellules, un processus décrit à la section §5.2.2.4. L'implémentation de l'assemblage de la matrice globale ne pose pas de problème particulier, car il s'agit avant tout d'un réarrangement de termes qui a été détaillé aux sections §3.3.1 pour la méthode d'Éléments Finis et §3.3.2, pour la méthode Galerkin discontinue.

Les matrices locales spécifiques à la méthode de Galerkin discontinue, i.e les matrices de *saut* $\mathbb{T}_K$ (3.47), de *pénalisation* $\mathbb{Y}_K$ (3.48), de *saut Dirichlet* $\mathbb{T}_{b,F}$ (3.49), et de *pénalisation Dirichlet* $\mathbb{Y}_{b,F}$ (3.50), n'ont pas été calculées à lors de la résolution des problèmes de cellules. Les quantités nécessaires, valeurs et flux aux faces, ont cependant été stockées. Lors de la résolution du problème grossier, l'instance *MPCube* peut accéder à l'ensemble de ces quantités, tous macroéléments compris, et calculer les matrices.

Une fois le problème matriciel assemblé, il est résolu par le biais d'un des solveurs de *MPCube* et la solution grossière C_H est sauvegardée.

Remarque 5.5 *Le seul véritable lien entre le problème grossier et le code de calcul* MPCube *est l'utilisation d'un des solveurs de* MPCube *pour la résolution. On dispose d'une autre gamme de solveurs* via *la librairie* NUMERICAL PLATON *[167], utilisée au sein de la maquette Python autonome décrite au chapitre §4.*

Toute cette partie de la chaîne de calcul SALOME-MPCube *peut donc être implémentée au sein d'une maquette indépendante de* MPCube. *On a cependant préféré l'y intégrer pour des raisons de cohérence, afin de minimiser le nombre de codes de calcul nécessaires à la chaîne* SALOME-MPCube.

5.2.3.2 Qualification de l'implémentation.

On valide dans cette section l'implémentation des objets qui assemblent et résolvent le problème grossier, c'est-à-dire les objets PROBLEME_GROSSIER_EFP1 et PROBLEME_GROSSIER_GAP1. Il ne s'agit pas ici de tester les méthodes multi-échelles Q_1/*VFDiam* et GD/*VFDiam* dans leurs intégralités, mais d'assurer que les

méthodes d'Éléments Finis et de Galrkin discontinue ont été correctement implémentées.

Pour cela, on suppose que la diffusivité D est égale à la matrice *Identité* et que le recouvrement est nul. On retrouve ainsi le cadre de l'Exemple 4.4 avec $(a,b) = (1,1)$ pour la dimension 2 et celui de l'Exemple 5.1 avec $(a,b,c) = (1,1,1)$ pour la dimension 3. Les fonctions $(\Phi_K^i)_{1\leq i\leq n}$ sont alors explicitement connues sur chaque macroélément, et on peut déterminer précisément tous les termes matriciels nécessaires aux deux schémas grossiers.

On résout donc plusieurs exemples du problème de diffusion suivant :

$$\left\{\begin{array}{rcll} -\Delta C & = & f & \text{dans } \Omega, \\ C & = & g_D & \text{sur } \Gamma_D, \\ D\nabla C\cdot\mathbf{n} & = & g_N & \text{sur } \Gamma_N, \end{array}\right. \tag{5.6}$$

où (Γ_D,Γ_N) forme une partition de $\Gamma = \partial\Omega$ la frontière du domaine Ω. $\mathbf{n}$ designe le vecteur normal unitaire extérieure au domaine. f, g_D et g_N sont respectivement le terme source, la condition aux limites de Dirichlet et la condition aux limites de Neumann.

Dans tout ce qui suit, on suppose que $\Omega =]0,1[^d$, où d est la dimension de travail. La discrétisation $T_H(\Omega)$ de Ω se compose de macroéléments carrés ou cubiques de longueur H.

Exemples en dimension 2**.**

Exemple 5.3 *Afin de tester l'implementation des conditions aux limites de Dirichlet, on considère dans cet exemple que :*

$$\begin{aligned} f &= 0, \\ \Gamma_D &= \partial\Omega, \\ g_D &= \sin(\pi x)\sinh(\pi y). \end{aligned}$$

La solution du problème est alors :

$$C = \sin(\pi x)\sinh(\pi y).$$

Exemple 5.4 *On note $\Gamma_{y,m}$ et $\Gamma_{y,p}$ les faces du domaine Ω pour lesquelles on a $\{y=0\}$ et $\{y=1\}$ respectivement. Afin de tester l'implémentation des conditions aux limites de Neumann, on considère dans cet exemple que :*

$$\begin{aligned} f &= 0, \\ \Gamma_N &= \Gamma_{y,m}\sqcup\Gamma_{y,p}, \\ g_N &= \frac{\partial}{\partial\mathbf{n}}\left(\sin(\pi x)\sinh(\pi y)\right), \\ \Gamma_D &= \partial\Omega\setminus\Gamma_N, \\ g_D &= \sin(\pi x)\sinh(\pi y).. \end{aligned}$$

La solution du problème est alors :

$$C = \sin(\pi x)\sinh(\pi y)$$

Exemple 5.5 *Afin de tester l'implémentation du terme source, on considère dans cet exemple que :*

$$\begin{aligned} f &= 2(x(1-x)+y(1-y)),\\ \Gamma_D &= \partial\Omega,\\ g_D &= 0. \end{aligned}$$

La solution du problème est alors :

$$C = x(1-x)y(1-y).$$

Le critère de validation choisit ici est l'erreur normalisée maximale entre valeurs calculées et théoriques aux noeuds $I \in \mathscr{I}$:

$$e_\infty(C_H) = \max_{I\in\mathscr{I}} \frac{|C_H(I) - C(I)|}{1+|C(I)|} \tag{5.7}$$

Il ne s'agit pas du maximum de l'erreur entre solution théorique et calculée car on ne recalcule pas la solution $C_{H,h}$ au niveau fin. De cette manière, la discrétisation des macroéléments K n'influence le résultat que par le biais des matrices de raideurs et de masses, influence que l'on néglige dans cette partie.

La Figure 5.12 présente l'évolution de l'erreur $e_\infty(C_H)$ en fonction de H lors de la résolution par la méthode d'Éléments Finis (cercle) et de Galerkin discontinue (carré) des exemples précédents : Exemple 5.3 (—○— et —□—), Exemple 5.4 (—○— et —□—) et Exemple 5.5 (—○— et —□—).

Que ce soit pour la méthode d'Éléments Finis ou celle de Galerkin discontinue, tous les exemples convergent en $\mathscr{O}(H^2)$ quand H tend vers 0. L'implémentation de ces méthodes est donc validée pour la dimension 2.

Exemples en dimension 3.

Exemple 5.6 *Afin de tester l'implementation des conditions aux limites de Dirichlet, on considère dans cet exemple que :*

$$\begin{aligned} f &= 0,\\ \Gamma_D &= \partial\Omega,\\ g_D &= \sin(\pi x)\sin(\pi y)\sinh(\sqrt{2}\pi z). \end{aligned}$$

La solution du problème est alors :

$$C = \sin(\pi x)\sin(\pi y)\sinh(\sqrt{2}\pi z).$$

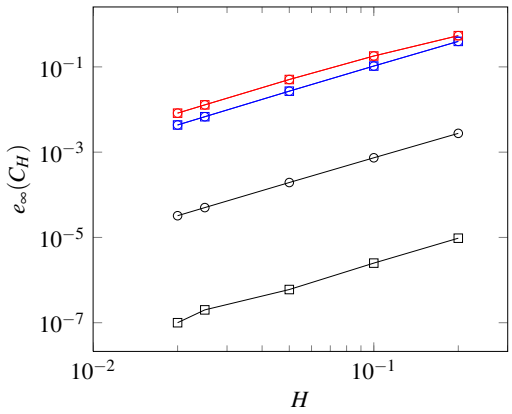

FIGURE 5.12 – Erreur $e_\infty(C_H)$ en fonction de H pour les Exemples 5.3 (—○—), 5.4 (—○—) et 5.5 (—○—) pour les méthodes d'Éléments Finis (—○—) et de Galerkin discontinue (—□—).

Exemple 5.7 *On note $\Gamma_{z,m}$ et $\Gamma_{z,p}$ les faces du domaine Ω pour lesquelles on a $\{z=0\}$ et $\{z=1\}$ respectivement. Afin de tester l'implémentation des conditions aux limites de Neumann, on considère dans cet exemple que :*

$$\begin{aligned} f &= 0, \\ \Gamma_N &= \Gamma_{z,m} \sqcup \Gamma_{z,p}, \\ g_N &= \frac{\partial}{\partial \mathbf{n}} \left(\sin(\pi x)\sin(\pi y)\sinh(\sqrt{2}\pi z) \right), \\ \Gamma_D &= \partial\Omega \setminus \Gamma_N, \\ g_D &= \sin(\pi x)\sin(\pi y)\sinh(\sqrt{2}\pi z). \end{aligned}$$

La solution du problème est alors :

$$C = \sin(\pi x)\sin(\pi y)\sinh(\sqrt{2}\pi z).$$

Exemple 5.8 *Afin de tester l'implémentation du terme source, on considère dans cet exemple que :*

$$\begin{aligned} f &= 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z), \\ \Gamma_D &= \partial\Omega, \\ g_D &= 0. \end{aligned}$$

La solution du problème est alors :

$$C = \sin(\pi x)\sin(\pi y)\sin(\pi z).$$

On utilise pour critère de validation le taux d'erreur définit en (5.7).

La Figure 5.13 présente l'évolution de l'erreur $e_\infty(C_H)$ en fonction de H lors de la résolution par la méthode d'Éléments Finis (cercle) et de Galerkin discontinue (carré) des des exemples précédents : Exemple 5.6 (—○— et —□—), Exemple 5.7 (—○— et —□—) et Exemple 5.8 (—○— et —□—).

Tous les exemples convergent en $\mathscr{O}(H^2)$ quand H tend vers 0, que ce soit pour la méthode d'Éléments Finis ou celle de Galerkin discontinue. L'implémentation de ces méthodes est donc validée pour la dimension 3.

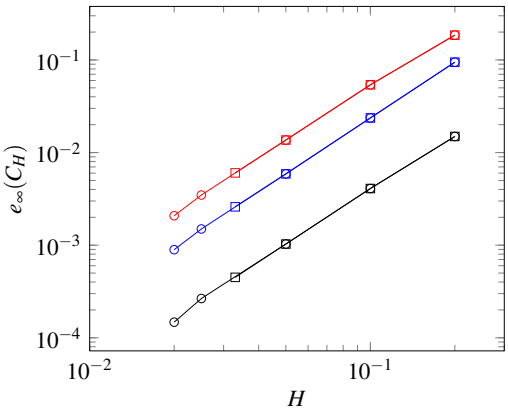

FIGURE 5.13 – Erreur $e_\infty(C_H)$ en fonction de H pour les Exemples 5.6 (—○—), 5.7 (—○—) et 5.8 (—○—) pour les méthodes d'Éléments Finis (—○—) et de Galerkin discontinue (—□—).