

TP : implémentation d'une méthode E.F.

Alexandre Vieira, Laurent Monasse
email: alexandre.vieira@inria.fr, laurent.monasse@inria.fr

13 février 2026

1 Objectifs et mise en place

Ce TP a pour objectif de vous faire pratiquer le développement effectif de la résolution de certaines EDP par une méthode d'éléments finis. Dans le cadre de ce TP, on ne s'intéressera qu'à la dimension 1 et aux éléments finis de Lagrange.

Organisation, méthodologie, évaluation Ce travail sera à faire par groupes de 4 personnes. Il vous sera demandé de rendre vos travaux pour (DATE À DÉFINIR). Une session de *code review* aura ensuite lieu le (DATE À DÉFINIR) où nous commenterons votre code et vous poserons des questions sur votre travail. Il est attendu que vous répartissiez les différentes tâches à faire entre membres du groupe, selon une méthodologie de travail qui vous sera demandé lors de la code review. Toute approche de méthodologie de gestion de projets (définition des rôles, répartition des tâches, méthode de suivi des avancées, utilisation d'outils de collaboration...) sera appréciée par vos évaluateurs. Si vous cherchez des pistes sur ce sujet, n'hésitez pas à venir en discuter après les TD.

Récupération du code Le code est stocké dans le répertoire git suivant : (LIEN GIT). Pour chaque groupe, il vous sera demandé de créer un fork et de travailler dans ce dernier. Une fois le fork créé, vous pourrez récupérer le code sur votre machine en utilisant la commande `git clone <lien git>`. Là encore, n'hésitez pas à venir en parler en TD si vous n'arrivez pas à récupérer le code.

Activation de l'environnement Le code est écrit en Julia et exploite les environnements afin de créer des codes portables entre différentes machines. Lors de vos développements, placez-vous depuis un terminal dans le répertoire 1D, puis lancez `julia`. Enfoncez alors] pour vous mettre en mode `pkg`, et activez l'environnement avec la commande "`activate .`" (le point est nécessaire). L'environnement de ce

répertoire sera alors actif. L'ensemble du package peut-être importé en utilisant `using FiniteElementSpace`.

Développement Pour le développement de ce projet, nous vous conseillons d'utiliser l'extension Julia que vous trouverez dans VSCode. Vous aurez ainsi accès à un interpréteur Julia intégré (avec la commande `Alt+J`, `Alt+O`) ainsi qu'une révision automatique du code à chaque modification enregistrée - ce qui vous permet de tout de suite tester vos modifications sans avoir à relancer l'interpréteur Julia et recompiler votre code.

Documentation Toutes les fonctions et structures du code sont documentées. Vous trouverez ces documentations directement dans le code, au-dessus des signatures, ou alors depuis l'interpréteur `julia` en tapant ? puis le nom de la fonction ou structure.

Détail des tâches demandées Nous allons vous demander différentes tâches, réparties en 3 thèmes. Ces thèmes seront détaillés dans les sections suivantes. Elles peuvent servir à vous répartir les tâches à faire, mais vous pouvez également décider de vous répartir autrement les tâches, du moment que vous gardez la charge de travail équitable entre les différents membres du groupe.

1. Une première tâche consistera à faire les liens nécessaires entre le maillage et les coefficients définissant une fonction éléments finis. Le travail ici consistera principalement à faire les liens corrects entre des numérotations locales (sur un élément de référence) et globales (sur le maillage). Les instructions se trouvent p.3
2. Une deuxième tâche consistera à construire les méthodes permettant de définir un élément fini puis un espace de fonctions définis par des éléments finis. Il faudra ici définir proprement les méthodes d'interpolation et comment exploiter les informations de l'élément de référence et du maillage. Les instructions commencent p.7
3. La dernière tâche concerne l'assemblage. Nous le ferons ici sur une équation de Helmholtz ($-u'' + u = f$). Il faudra ici assembler à la fois la matrice globale et inclure les conditions au bord dans la résolution des équations. Les instructions commencent p.12

À travers les différentes tâches, nous vous demanderons de développer également des tests. Les tests unitaires sont à ajouter dans le répertoire `test` afin de pouvoir être lancés automatiquement (voir les explications p.18). Les tests fonctionnels (pour la dernière partie) sont à développer dans des fichiers de scripts à la racine du projet (comme pour `launch_Helmholtz.jl`).

2 Gestion de la numérotation des éléments et du maillage

Votre travail consistera dans cette partie à identifier les nœuds (ou points) présents dans l'élément de référence et faire le lien avec les noeuds présents dans le maillage.

2.1 Définition du maillage

Dans le cas 1D, un maillage est défini comme un ensemble de points et d'arêtes liant ces points. Celui-ci est défini par 2 tableaux :

- Un premier tableau, `vertices`, définit l'ensemble des extrémités des segments.
- Un deuxième tableau, `edges`, définit l'ensemble des segments du maillage. Il s'agit d'un tableau de connectivité orienté : chaque colonne représente un segment ; toute la colonne vaut 0 sauf en deux lignes, valant -1 pour le point de départ du segment et 1 pour le point d'arrivée.

Un exemple est donné figure 1.

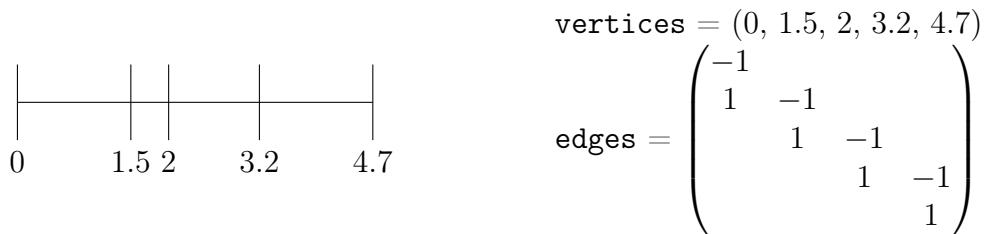


FIGURE 1 – Exemple d'encodage d'un maillage 1D.

Tâche 1 Complétez la fonction `generate_mesh_structured` dans le fichier `Mesh1D.jl`. Cette fonction doit prendre en entrée les extrémités d'un intervalle $[a, b]$ ainsi que le nombre de sous-intervalles, et renvoie un `Mesh1D` encodant le maillage associé. Vous devrez également remplir le tableau `entity_counts` en accord avec la documentation de `Mesh1D`. La fonction `spzeros` vous permet d'initialiser un tableau sparse de type `SparseMatrixCSC`; allez voir sa documentation.

Test 1 Écrivez deux tests unitaires permettant de vous assurer que les maillages sont bien construits.

2.2 Définition des nœuds dans l'élément de référence

La méthode des éléments finis se base sur une intégration effectuée sur un élément de référence. Ici, on prendra comme élément de référence l'intervalle $[-1, 1]$, même si le code autorise d'autres intervalles de référence (à prendre en compte dans votre code!). Sur cet élément de référence, on définit un ensemble de nœuds équirépartis servant à définir les polygones de base servant à définir la base de l'espace fonctionnel sur lequel on résoud les EDP. Ce sont en ces nœuds qu'on définit les polynomes suivant la règle $\hat{P}_i(x_j) = \delta_{ij}$. Nous avons déjà vus certains exemples en TD :

- L'espace des polynomes de degré 1 est de dimension 2 (les deux coefficients dans $ax + b$), il faut donc 2 degrés de libertés et donc deux nœuds sur l'élément de référence, situés aux extrémités de l'intervalle.
- L'espace des polynomes de degré 2 est de dimension 3 (les trois coefficients dans $ax^2 + bx + c$), il faut donc 3 degrés de libertés et donc trois nœuds sur l'élément de référence : deux aux extrémités, un au milieu de l'intervalle.

On voit qu'ainsi, pour définir un espace de polynomes de degré k (noté \mathbb{P}_k), il nous faut $k + 1$ nœuds sur l'élément de référence. En les supposant équirépartis sur l'intervalle, on peut facilement calculer la position de chaque nœud dans l'intervalle. Ceci est illustré dans la figure 2.

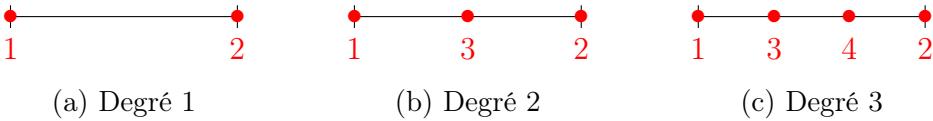


FIGURE 2 – Position des noeuds, ainsi que leur numérotation, dans l'élément de référence suivant le degré de l'élément finis.

On définit, par la même occasion, une numérotation sur les nœuds. Pour des raisons pratiques lors de la numérotation globale, ces nœuds sont numérotés par entité (une entité étant soit un point, de dimension 0, soit une arête, de dimension 1). Ainsi, on commence par numéroter les nœuds associés aux points (les extrémités de l'intervalle) puis on numérote les nœuds à l'intérieur de l'intervalle, associés à l'arrête.

Tâche 2 Implémentez la génération des nœuds de Lagrange dans la fonction `LagrangeElement1D` dans `FiniteElement.jl`. Cela se fait en une ligne avec les fonctions `range` et `collect`. Laissez de côté l'attribut `basis_coeffs`, rempli par un de vos collègues. Implémentez également la construction du dictionnaire `entity_nodes` donnant la liste des nœuds associés à chaque entité. Regardez bien la structure de ce dictionnaire ainsi que la documentation de

`FiniteElement1D`. En reprenant la numérotation dans la figure 2, vous devriez obtenir :

- Degré 1 : `entity_nodes[0] = [1 => [1], 2 => [2]]`,
`entity_nodes[1] = []`
- Degré 2 : `entity_nodes[0] = [1 => [1], 2 => [3]]`,
`entity_nodes[1] = [1 => [2]]`
- Degré 3 : `entity_nodes[0] = [1 => [1], 2 => [4]]`,
`entity_nodes[1] = [1 => [2,3]]`

Test 2 Écrivez au moins deux tests unitaires permettant de vous assurer que les `FiniteElement1D` sont bien construits. Pensez à tester plusieurs configurations pour tester chaque attribut que vous générez dans la tâche 2.

2.3 Numérotation globale

Dans la section précédente, nous avons défini une numérotation locale pour chaque noeud. Chaque noeud définit un coefficient dans l'approximation éléments finis d'une fonction u ; par exemple, en \mathbb{P}_2 , on définit une fonction sur l'élément E_k par $u(x) = u_0^k P_0(x) + u_m^k P_m(x) + u_1^k P_1(x)$. Les coefficients u_i^k sont définis élément par élément (ou en 1D, sous-intervalle par sous-intervalle), certains de ces coefficients étant communs à deux éléments (pensez aux coefficients associés à l'extrémité commune à deux sous-intervalles). Afin de distinguer chaque coefficient dans le maillage, la numérotation locale ne suffit plus. On doit donc définir une numérotation globale de ces coefficients, dans l'ensemble du maillage mais qui est basée sur la numérotation locale dans l'élément de référence (et notamment, sur l'organisation par entité).

Il existe plusieurs numérotations possibles (qui peuvent être optimisées pour des raisons de performances de calcul, afin de limiter les sauts dans la mémoire). Nous allons ici rester sur une numérotation simple. Comme on se base sur la numérotation locale, on va garder la même organisation : on commence par numérotter tous les noeuds associés aux points du maillage (la dimension 0) puis on numérote les noeuds associés aux arêtes. Cette logique donne par exemple la numérotation illustrée Figure 3 (voyez la correspondance qu'on peut y voir avec le degré 2 dans la Figure 2).

On a donc une formule qui permet de calculer le premier indice associé à l'entité de dimension d pour le sous-intervalle i :

$$G_{d,i} = \left(\sum_{\delta < d} N_\delta E_\delta \right) + i N_d$$

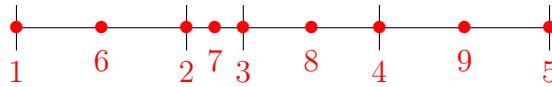


FIGURE 3 – Numérotation globale, dans un cas d’EF de degré 2. Dans la cellule 3, le nœud d’indice local 2 devient le nœud d’indice global 8 (ou écrit autrement, l’indice local (3,2) devient l’indice global 8). Le nœud d’indice global 4 est à la fois l’indice local (3,3) et l’indice local (4,1).

où N_d est le nombre de nœuds associé à chaque entité de dimension d , et E_d le nombre d’entité de dimension d dans le maillage. Dans l’exemple de la figure 3, on aurait donc $N_0 = 1$, $N_1 = 1$, $E_0 = 5$, $E_1 = 4$.

Grâce à cette construction, il existe une correspondance entre les nœuds définis sur l’élément de référence et les nœuds défini sur chaque sous-intervalle. Cette correspondance est stockée dans un tableau de dimension 2 de taille (nombre de sous-intervalles \times nombre de nœud par sous-intervalle). Dans chaque ligne, on respecte l’ordre dans lequel sont rangés les nœuds dans l’élément de référence : on liste d’abord les indices liés aux extrémités du sous-intervalle (entité de dimension 0), puis on liste les indices liés aux arêtes (entité de dimension 1). Ainsi, pour l’exemple dans la figure 3, on devrait avoir le tableau suivant :

$$\text{cell_node_mappings} = \begin{pmatrix} 1 & 2 & 6 \\ 2 & 3 & 7 \\ 3 & 4 & 8 \\ 4 & 5 & 9 \end{pmatrix}$$

Tâche 3 Terminez la définition du constructeur `FiniteElementSpace1D` dans `FiniteElementSpace.jl`. Le travail consiste principalement à compléter la définition de `cell_node_mappings`. Vous pourriez pour cela vous intéresser à la fonction `cumsum` de Julia ainsi qu’à la fonction `adjacency` définie dans `Mesh1D.jl` qui renvoie les indices des points associés à l’arête numéro i (voir la documentation). Attention : la structure de données ne suppose pas forcément que les points et les arêtes sont rangés dans l’ordre croissants !

Test 3 Écrivez au moins un test unitaire permettant de vous assurer que `cell_node_mappings` est bien construit.

3 Définition des fonctions éléments finis

Dans cette partie, nous allons définir les méthodes permettant de construire une approximation éléments finis d'une fonction. Pour rappel, on construit un sous-espace vectoriel de fonctions en se donnant des fonctions polynomiales par morceaux, de degré k et supportés sur des sous-intervalles $[x_i, x_{i+1}]$, qu'on note $P^{i,j}$. Par suite, on appelle approximation E.F. \mathbb{P}_k les fonctions de la forme $u(x) = \sum_{i,j} u_{i,j} P^{i,j}(x)$.

Comme on l'a vu en cours, les $P^{i,j}$ sont définis à partir de polynômes \hat{P}^j défini sur un élément de référence \hat{E} . Et comme on l'a également vu en TD, ces \hat{P}^j peuvent être définis à partir de nœuds $\{\xi_k\}$ dans l'élément de référence à partir desquels on construit les polynômes \hat{P}^j de sorte que $\hat{P}^j(\xi_k) = \delta_{jk}$. Cette dernière égalité nous permet de définir tous les coefficients des monômes des \hat{P}^j . On se propose ici d'en déduire un algorithme permettant de définir ces \hat{P}^j .

3.1 Matrice de Vandermonde

Avant de généraliser, on va commencer par un exemple. Sur un élément de référence $\hat{E} = [a, b]$, on définit une base de nœuds $\hat{\xi}_0 = a$, $\hat{\xi}_m = \frac{1}{2}(a + b)$, $\hat{\xi}_1 = b$. À partir de ces trois nœuds, on peut définir 3 polynômes \hat{P}^j de degré deux tels que $\hat{P}^j(\hat{\xi}_k) = \delta_{ij}$. Notez que l'ensemble de polynômes de degré 2 étant de dimension 3 (il y a 3 coefficients dans $ax^2 + bx + c\dots$), avoir 3 points dans \hat{E} suffit.

Pour $j = 0$, l'égalité $\hat{P}^j(\hat{\xi}_k) = c_j \hat{\xi}_k^2 + b_j \hat{\xi}_k + a_j = \delta_{kj}$ s'écrit sous forme de système :

$$\begin{pmatrix} c_0 \hat{\xi}_0^2 + b_0 \hat{\xi}_0 + a_0 \\ c_0 \hat{\xi}_1^2 + b_0 \hat{\xi}_1 + a_0 \\ c_0 \hat{\xi}_2^2 + b_0 \hat{\xi}_2 + a_0 \end{pmatrix} = \begin{pmatrix} 1 & \hat{\xi}_0 & \hat{\xi}_0^2 \\ 1 & \hat{\xi}_1 & \hat{\xi}_1^2 \\ 1 & \hat{\xi}_2 & \hat{\xi}_2^2 \end{pmatrix} \begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

À partir de là, les coefficients apparaissent de façon évidente :

$$\begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix} = \begin{pmatrix} 1 & \hat{\xi}_0 & \hat{\xi}_0^2 \\ 1 & \hat{\xi}_1 & \hat{\xi}_1^2 \\ 1 & \hat{\xi}_2 & \hat{\xi}_2^2 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

On peut généraliser à tout j , ce qui donne le système matriciel à résoudre :

$$\begin{pmatrix} 1 & \hat{\xi}_0 & \hat{\xi}_0^2 \\ 1 & \hat{\xi}_1 & \hat{\xi}_1^2 \\ 1 & \hat{\xi}_2 & \hat{\xi}_2^2 \end{pmatrix} \begin{pmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ c_0 & c_1 & c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Vous reconnaissiez sans doute la matrice de Vandermonde à gauche. Ce qui a été développé ici pour des polynômes de degré 3 peut aisément se généraliser à

des polynômes de degré n en se basant sur $n + 1$ points en utilisant la matrice de Vandermonde :

$$V = \begin{pmatrix} 1 & \hat{\xi}_0 & \cdots & \hat{\xi}_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \hat{\xi}_n & \cdots & \hat{\xi}_n^n \end{pmatrix}$$

Ainsi, la matrice C des coefficients définissant les polynômes \hat{P}^j est définie par $C = V^{-1}$, où C_{kj} est le coefficient du k -ème monôme (devant ξ^{k-1}) de \hat{P}^j .

Tâche 1 Implémentez la fonction `VandermondeMatrix` dans `FiniteElement.jl`.

Ignorez pour le moment l'argument `grad` et considérez le comme `false`. Une fois qu'un ou une de vos collègues aura finit la tâche 2 de la section 2, terminez d'implémenter `LagrangeElement1D` en ajoutant l'expression de `basis_coeffs` contenant les coefficients des polynômes de base sur un élément fini.

Test 1 Écrivez au moins deux tests unitaires permettant de vous assurer que `VandermondeMatrix` est bien construite.

Remarque Pour votre culture mathématique, sachez que cette approche basée sur les matrices de Vandermonde construites avec les monomes n'est pas utilisée en pratique. En effet, la matrice construite ainsi a un conditionnement important à mesure qu'on augmente le degré des polynômes. Les méthodes réellement utilisées restent en dehors du programme de ce cours.

3.2 Évaluation de la base polynomiale

Dans ce code, nous allons avoir besoin de calculer de façon intensive des intégrales. Ces intégrales seront calculées par des formules de quadrature (ça tombe bien, on n'a que des polynômes). Et ces formules de quadrature nécessitent qu'on évalue des fonctions polynomiales en certains points donnés. Le fait d'évaluer un ensemble de fonctions en un ensemble de points donnés s'appelle *tabuler*, et ce ce que nous allons coder ici.

Ici, on va vouloir tabuler la base polynomiale $\{\hat{P}^j\}$ avec un ensemble de points $\{\xi_i\}$ (qui ne sont pas nécessairement les noeuds $\hat{\xi}_i$ utilisés pour définir les \hat{P}^j !). Si on note $T_{ij} = \hat{P}^j(\xi_i)$, on obtient :

$$T_{ij} = \sum_{k=1}^{\deg(\hat{P}^j)} C_{kj} \xi_i^{k-1} = \sum_{k=1}^{\deg(\hat{P}^j)} C_{kj} V(\xi_i)_k = (V(\xi_i) \times C)_{ij}. \quad (1)$$

Tâche 2 Implémentez la fonction `evaluate_basis` dans `FiniteElement.jl` en utilisant (1), en ignorant encore l'argument `grad`. Cela peut se faire en littéralement une ligne...

3.3 Construction des polynômes dérivés

Les polynômes que nous avons construits précédemment (à travers leurs coefficients) sont utilisés par la suite pour construire les matrices globales dans la résolution des EDP. Cependant, les formulations faibles font parfois apparaître des dérivées, ce qui demande de calculer également les dérivées des polynômes \hat{P}^j . Revenons à la définition des fonctions E.F. définie sur un seul élément : $u(x) = \sum_j u_j \hat{P}^j(x)$. Si on veut calculer leurs dérivées, on doit donc avoir $u'(x) = \sum_j u_j (\hat{P}^j)'(x)$. Il faut donc également tabuler les $(\hat{P}^j)'$. Or, en revenant à la définition de T_{ij} dans (1), on voit que le tableau $(T'_{ij})_{ij} = \left(\frac{d\hat{P}^j}{dx}(\xi_i) \right)_{ij}$ s'exprime comme :

$$T' = \nabla_\xi (V(\xi:) \times C) = (\nabla V(\xi:)) \times C \quad (2)$$

On doit donc construire la matrice ∇V définie comme $(\nabla V(\xi:))_{ij} = \frac{dV_j}{d\xi}(\xi_i)$ (où pour rappel, $V_k(\xi_i) = (\xi_i)^{k-1}$, le k -ème monôme).

Tâche 3 Complétez la fonction `VandermondeMatrix` dans `FiniteElement.jl` en utilisant (2) pour le cas où `grad = true`. Assurez-vous que `evaluate_basis` fonctionne toujours avec `grad = true`.

Test 2 Écrivez au moins deux tests unitaires permettant de vous assurer que `evaluate_basis` fonctionne correctement.

3.4 Interpolation dans l'élément de référence

On reste pour le moment dans un élément de référence. En reprenant l'expression $u(x) = \sum_j u_j \hat{P}^j(x)$, on remarque que comme $\hat{P}^j(\hat{\xi}_i) = \delta_{ij}$, on a de suite que $u(\hat{\xi}_i) = \sum_j u_j \delta_{ij} = u_i$.

Tâche 4 Implémentez la fonction `interpolate` dans `FiniteElement.jl`. Cela peut se faire en littéralement une ligne en utilisant la fonction de Julia qui s'appelle `map`.

3.5 Interpolation dans le maillage complet

De la même façon que l'interpolation sur l'élément de référence se fait simplement avec des évaluations de fonctions, l'interpolation sur le maillage entier se fera aussi avec des évaluations de fonctions. Pour rappel, à partir de l'expression $u(x) = \sum_{i,j} u_{i,j} P^{i,j}(x)$, on remarque qu'en utilisant le fait que $P^{i,j}(x_{i,j}) = \delta_{ij}$, on a directement $u(x_{i,j}) = u_{i,j}$. Il nous faut donc un moyen d'obtenir les nœuds $x_{i,j}$. Dit autrement, il nous faut passer de l'élément de référence (les ξ_i) à l'élément physique j (les $x_{i,j}$). Voir la figure 4 pour une illustration.

Pour cela, plaçons-nous sur l'élément de référence $[-1, 1]$. Dans cet élément, les fonctions de base dans \mathbb{P}_1 sont $\hat{P}_e^0(\xi) = \frac{1}{2}(1 - \xi)$ et $\hat{P}_e^1(\xi) = \frac{1}{2}(1 + \xi)$. Prenons un nœud $\xi_i \in [-1, 1]$. Par définition de l'appartenance à un intervalle, il existe $\lambda_i \in [0, 1]$ tels que $\xi_i = \lambda_i(-1) + (1 - \lambda_i)(1)$ (on appelle λ_i la coordonnée barycentrique de ξ_i dans $[-1, 1]$). Prenons par exemple $\xi_m = 0$, le milieu de l'intervalle $[-1, 1]$. Sa coordonnée barycentrique est $\lambda_m = \frac{1}{2}$.

Prenons maintenant un élément physique $[a, b]$. Au nœud ξ_i dans $[-1, 1]$, on associe un nœud x_i dans $[a, b]$ avec la même coordonnée barycentrique λ_i , c'est-à-dire $x_i = \lambda_i a + (1 - \lambda_i)b$. Si on reprend $\xi_m = 0$, il est associé au point $x_m = \frac{1}{2}a + \frac{1}{2}b$, qui est le milieu de $[a, b]$.

Toute la *magie* de cette approche est qu'on peut calculer les λ_i à partir de \hat{P}_e^0 . En effet, remarquez d'abord qu'on a $\hat{P}_e^0(\xi) = 1 - \hat{P}_e^1(\xi)$ et que $\lambda_i = \hat{P}_e^0(\xi_i)$. À partir de cette observation, on remarque que $x_i = \hat{P}_e^0(\xi_i)a + \hat{P}_e^1(\xi_i)b = \sum_{j=0}^1 \hat{x}_i \hat{P}_e^j(\xi_i)$ où \hat{x}_i sont les extrémités de $[a, b]$ (donc $\hat{x}_0 = a$ et $\hat{x}_1 = b$). Si on reprend l'exemple du point milieu $\xi_m = 0$, $\hat{P}_e^0(\xi_m) = \frac{1}{2} = \lambda_m$. Puis on retrouve bien $x_m = \sum_{j=0}^1 \hat{x}_i \hat{P}_e^j(\xi_m) = \frac{1}{2}a + \frac{1}{2}b$.

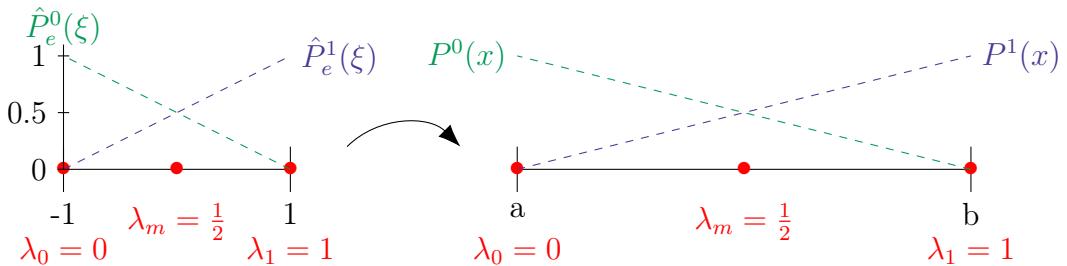


FIGURE 4 – Lien sur les barycentres entre élément de référence et élément physique.

Avec toutes ces observations, on peut décrire l'algorithme suivant pour l'interpolation d'une fonction u sur une base E.F. de degré k :

1. On construit une approximation E.F. de degré k d'une fonction u avec tous les coefficients nuls.

2. On construit un élément fini de degré 1 sur l'élément de référence. Avec les polynômes \mathbb{P}_1 définis ainsi, on calcule les coordonnées barycentriques $\hat{P}_e^j(\xi_i)$, $j = 0, 1$ des $k+1$ nœuds ξ_i dans l'élément de référence. Cela forme une matrice λ_array de taille $(k + 1, 2)$ composée de tous les λ_i et $1 - \lambda_i$ (matrice qu'on peut calculer avec la fonction `evaluate_basis`).
3. Pour chaque sous-intervalle dans le maillage :
 - On récupère les coordonnées du bord de chaque sous-intervalle.
 - On calcule les positions des nœuds x_i dans le sous-intervalle à partir des coordonnées barycentriques.
 - On calcule les coefficients $u(x_i)$ pour chaque nœud x_i .

Tâche 5 Complétez la fonction `interpolate` dans `FiniteElementSpace.jl`. Une partie de la fonction est déjà implémentée en commentaire, vous devez compléter ce qu'il manque. Vous aurez pour cela besoin de comprendre comment s'organise la numérotation des nœuds entre l'élément de référence et le sous-intervalle courrant (ou l'élément physique) : pour cela, échangez avec votre collègue s'occupant du maillage.

Test 3 Écrivez au moins deux tests unitaires permettant de vous assurer que `interpolate` est bien construite.

4 Assemblage de la matrice, résolution du problème E.F.

Dans cette partie, nous allons nous concentrer sur l'assemblage du système linéaire à résoudre nous permettant d'approximer la résolution d'EDP. Pour cela, on ne s'intéressera qu'au problème de Helmholtz :

$$-u'' + u = f, \quad \text{sur }]a, b[$$

avec soit des conditions de Dirichlet ou de Neumann au bord. Cette équation est résolue au sens faible :

$$\int_a^b u'v' + uv = \int_a^b fv, \forall v \in V, \quad (3)$$

avec $V = H^1([a, b])$ si on a des conditions de Neumann en a et b ou $H_0^1([a, b])$ si on a des conditions de Dirichlet.

On va approximer la solution de cette équation par une méthode éléments finis. Pour cela, on se donne un maillage de $[a, b]$ constitué de $N_E + 1$ sous-intervalles $[x_i, x_{i+1}]$, $i = 0, \dots, N_E$. Sur chaque sous-intervalle $[x_i, x_{i+1}]$, on définit une base de polynômes P_i^j , $j = 0, \dots, N_P$ de degré N_P . La solution u est approchée par une fonction u_h dans un s.e.v. V_h généré par les P_i^j ; dit autrement, on a $u_h \in V_h$ si pour tout i , il existe des $u_{i,j} \in \mathbb{R}$ tels que $u_h(x) = \sum_{j=0}^{N_P} u_{i,j} P_i^j(x) \forall x \in [x_i, x_{i+1}]$. Les indices locaux (i, k) , qui désignent le noeud k dans la cellule i , sont globalisés dans le maillage : vous pouvez regarder la figure 3 et demander à votre collègue s'occupant du maillage de vous expliquer.

Si on remplace V par V_h dans (3), on obtient à gauche :

$$\begin{aligned} \int_a^b u'_h v'_h + u_h v_h &= \sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} \sum_{j=0}^{N_P} \sum_{k=0}^{N_P} (u_{i,j}(P_i^j)') \sum_{k=0}^{N_P} (v_{i,k}(P_i^k)') + (u_{i,j} P_i^j) \sum_{k=0}^{N_P} (v_{i,k} P_i^k) \\ &= \sum_{j=0}^{N_P} \sum_{k=0}^{N_P} \left(u_{i,j} v_{i,k} \left(\sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} ((P_i^j)'(P_i^k)' + P_i^j P_i^k) \right) \right). \end{aligned}$$

À droite, on obtient de la même manière :

$$\int_a^b fv = \sum_{k=0}^{N_P} v_{i,k} \left(\sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} f P_i^k \right)$$

Si on définit la matrice $A_{(i,k)(i,j)} = \sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} ((P_i^j)'(P_i^k)' + P_i^j P_i^k)$ et le vecteur $\mathbf{b}_{(i,k)} = \sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} f P_i^k$, on voit que l'égalité (3) dans V_h devient, en posant $\mathbf{u} = \{u_{i,j}\}_{(i,j)}$, $\mathbf{v} = \{v_{i,k}\}_{(i,k)}$:

$$\langle A\mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{b}, \mathbf{v} \rangle, \quad \forall \mathbf{v} \in \mathbb{R}^{N_P},$$

ce qui équivaut à devoir résoudre le problème linéaire $A\mathbf{u} = \mathbf{b}$.

Tout l'objectif de cette section va être de construire de façon efficace la matrice A et le vecteur \mathbf{b} .

4.1 Intégration numérique

Une grande partie du travail consistera à calculer des intégrales. Ces intégrales seront calculées (ou approximées) par des méthodes de quadrature et un changement de variable.

Supposons qu'on souhaite calculer l'intégrale sur un intervalle $[x_i, x_{i+1}]$ d'une fonction f . On va pour cela se ramener à un élément de référence $[\xi_0, \xi_1]$ en passant par une application $\tau_i : [\xi_0, \xi_1] \rightarrow [x_i, x_{i+1}]$ et faire un changement de variable :

$$\int_{x_i}^{x_{i+1}} f(x) dx = \int_{\xi_0}^{\xi_1} f(\tau_i(\xi)) \tau'_i(\xi) d\xi.$$

Or, comme on l'a vu en TD, τ_i est une application affine. Sa dérivée est donc une constante qui vaut $\tau'_i(\xi) = \frac{x_{i+1}-x_i}{\xi_1-\xi_0} = \tau'_i$. Ainsi, on doit simplement calculer :

$$\int_{x_i}^{x_{i+1}} f(x) dx = \tau'_i \int_{\xi_0}^{\xi_1} f(\tau_i(\xi)) d\xi.$$

Supposons maintenant que f soit une approximation éléments finis \mathbb{P}_k , c'est-à-dire que sur $[x_i, x_{i+1}]$, il existe des fonctions polynomiales P_i^j tels que $f(x) = \sum_{j=0}^k f_j P_i^j(x)$, $\forall x \in [x_i, x_{i+1}]$. Comme on l'a vu en TD, ces P_i^j sont définis à travers l'élément de référence $[\xi_0, \xi_1]$, sur lequel on définit des polynômes \hat{P}^j , puis on pose $P_i^j = \hat{P}^j \circ (\tau_i)^{-1}$ (ou de la même façon, $\hat{P}^j = P_i^j \circ \tau_i$). En utilisant cela, le calcul de l'intégrale devient :

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x) dx &= \tau'_i \int_{\xi_0}^{\xi_1} \sum_{j=0}^k f_j P_i^j(\tau_i(\xi)) d\xi \\ &= \sum_{j=0}^k \tau'_i f_j \int_{\xi_0}^{\xi_1} P_i^j(\tau_i(\xi)) d\xi \\ &= \sum_{j=0}^k \tau'_i f_j \int_{\xi_0}^{\xi_1} \hat{P}^j(\xi) d\xi. \end{aligned}$$

Il ne reste donc plus qu'à calculer l'intégrale des \hat{P}^j ! Cela se fait avec une règle de quadrature : on évalue \hat{P}^j en un ensemble de points ξ_ℓ avec des poids ω_ℓ , pour $\ell = 0, \dots, d$ où d est le degré de la quadrature. Pour que la quadrature soit

exacte pour les polynômes \hat{P}^j de degré k , on doit prendre $d = k$. Ainsi, le calcul de l'intégrale devient :

$$\int_{x_i}^{x_{i+1}} f(x) dx = \sum_{j=0}^k \sum_{\ell=0}^k \tau'_i f_j \omega_\ell \hat{P}^j(\xi_\ell). \quad (4)$$

Tâche 1 Écrivez la fonction `jacobian` dans `Mesh1D.jl`, qui retourne la valeur de τ'_i pour le sous-intervalle $i = \text{i_cell}$. Vous aurez besoin pour cela de la fonction `adjacency` (regardez sa documentation). Pensez à vous assurer que $x_{i+1} - x_i > 0$ (ou utilisez un `abs`). Étant donné que la valeur $\xi_1 - \xi_0$ est la même pour tous les sous-intervalle (cela ne dépend que de l'élément de référence qui est toujours le même), vous pouvez précalculer cette valeur dans l'attribut `scaling` dans le constructeur de `Mesh1D`.

Tâche 2 Complétez la fonction `integrate` dans `FiniteElementSpace.jl`. Vous pouvez vous inspirer que la fonction `errornorm` dans le même fichier, qui calcule $\|f_1 - f_2\|_{L^2}$ (et donc, calcule une intégrale...).

Test 1 Écrivez au moins un test unitaire permettant de vous assurer que `integrate` fonctionne correctement.

4.2 Assemblage du second membre \mathbf{b}

On repart de l'expression de $\mathbf{b}_{(i,k)} = \sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} f P_i^k$. On notera N l'indice global correspondant à l'indice local (i, k) (voir encore la figure 3). Sur chaque sous-intervalle $[x_i, x_{i+1}]$, on décompose f sur la base de polynômes P_i^j : $\forall x \in [x_i, x_{i+1}]$, $f(x) = \sum_{j=0}^k f_j^i P_i^j(x)$. En l'introduisant dans \mathbf{b}_N , on obtient :

$$\mathbf{b}_N = \sum_{i=0}^{N_E} \sum_{j=0}^{N_P} f_j^i \int_{x_i}^{x_{i+1}} P_i^j P_i^k.$$

L'intégrale $\int_{x_i}^{x_{i+1}} P_i^j P_i^k$ sera calculée comme dans (4) en utilisant une quadrature de degré $2N_P$ (qui sera exacte, vu que les P_i^j sont des polynômes de degré N_P !). Ainsi on aura :

$$\mathbf{b}_N = \sum_{i=0}^{N_E} \left[\sum_{j=0}^{N_P} f_j^i \int_{x_i}^{x_{i+1}} P_i^j P_i^k \right] = \sum_{i=0}^{N_E} \left[\sum_{\ell=0}^{2N_P} \tau'_i \omega_\ell \hat{P}^k(\xi_\ell) \left(\sum_{j=0}^{N_P} f_j^i \hat{P}^j(\xi_\ell) \right) \right]$$

Avec cette formule, un algorithme se dégage : on va parcourir les sous-intervalles les uns après les autres, et ajouter au fur et à mesure les contributions entre crochets aux \mathbf{b}_N . L'algorithme est décrit comme suit :

1. On part d'un vecteur nul $\mathbf{b} = 0$.
2. Pour chaque sous-intervalle c :
 - (a) Pour chaque noeud i dans le sous-intervalle
 - i. Trouver dans la numérotation globale l'indice N correspondant au noeud i dans le sous-intervalle c .
 - ii. Pour tout $\ell = 0, \dots, N_P$, on calcule $g_\ell = \sum_{j=0}^{N_P} f_j^c \hat{P}^j(\xi_\ell)$
 - iii. On calcule $Z = \sum_{\ell=0}^{2N_P} \omega_\ell \hat{P}^i(\xi_\ell) g_\ell$.
 - iv. On l'ajoute à \mathbf{b}_N : $\mathbf{b}_N = \mathbf{b}_N + \tau'_c Z$

Tâche 3 Complétez la fonction `build_rhs` dans `Helmholtz.jl`. Identifiez pour cela comment calculer les fonctions $\hat{P}^j(\xi_\ell)$ avec la fonction `evaluate_basis`. Les g_ℓ peuvent se calculer avec un produit matrice/vecteur. Z se calcule comme un produit scalaire avec un produit terme à terme de vecteurs. Un produit scalaire se calcule avec `dot` (dans la bibliothèque `LinearAlgebra`), un produit terme à terme se calcule en utilisant `.*` entre deux vecteurs.

4.3 Assemblage de la matrice A

De la même façon, on va construire la matrice A sous-intervalle par sous-intervalle. Ici, on a directement l'expression $A_{(i,k)(i,j)} = \sum_{i=0}^{N_E} \int_{x_i}^{x_{i+1}} ((P_i^j)'(P_i^k)' + P_i^j P_i^k)$. À nouveau, on se ramène au calcul sur l'élément de référence $[\xi_0, \xi_1]$. Cependant, il faut faire attention avec les dérivées : comme on l'a vu en TD, l'intégration devient

$$\begin{aligned} \int_{x_i}^{x_{i+1}} (P_i^j)'(x)(P_i^k)'(x) dx &= \int_{\xi_0}^{\xi_1} \frac{(\hat{P}_i^j)'(\xi)}{\tau'_i(\xi)} \frac{(\hat{P}_i^k)'(\xi)}{\tau'_i(\xi)} \tau'_i(\xi) d\xi \\ &= \frac{1}{\tau'_i} \int_{\xi_0}^{\xi_1} (\hat{P}_i^j)'(\xi)(\hat{P}_i^k)'(\xi) d\xi. \end{aligned}$$

Ainsi, en utilisant encore une fois une formule de quadrature, on définit :

$$a'_{ijk} = \int_{x_i}^{x_{i+1}} (P_i^j)'(x)(P_i^k)'(x) dx = \sum_{\ell=0}^{2N_P} \frac{\omega_\ell}{\tau'_i} (\hat{P}^j)'(\xi_\ell)(\hat{P}^k)'(\xi_\ell).$$

De même, on définit :

$$a_{ijk} = \int_{x_i}^{x_{i+1}} P_i^j(x) P_i^k(x) dx = \sum_{\ell=0}^{2N_P} \tau'_i \omega_\ell \hat{P}^j(\xi_\ell) \hat{P}^k(\xi_\ell).$$

On notera N l'indice global correspondant à (i, k) et M l'indice global correspondant à (i, j) . Ainsi on a établit la formule :

$$A_{NM} = \sum_{i=0}^{N_E} (a'_{ijk} + a_{ijk}).$$

L'algorithme se développe de façon analogue à **b** :

1. On part d'une matrice nulle $A = 0$.
2. Pour chaque sous-intervalle c :
 - (a) Pour chaque noeud i dans le sous-intervalle
 - i. Trouver dans la numérotation globale l'indice N correspondant au noeud i dans le sous-intervalle c .
 - ii. Pour chaque noeud j dans le sous-intervalle
 - A. Trouver dans la numérotation globale l'indice M correspondant au noeud j dans le sous-intervalle c .
 - B. Calculer a_{cij} .
 - C. Calculer a'_{cij}
 - D. On l'ajoute à A_{NM} : $A_{NM} = A_{NM} + a_{cij} + a'_{cij}$

Tâche 4 Complétez la fonction `build_lhs` dans `Helmholtz.jl`. Remarquez que vous pouvez calculer en amont les produits $\hat{P}^j(\xi_\ell)\hat{P}^k(\xi_\ell)$ qui ne dépendent pas de la cellule c . Vous pouvez faire de même avec les produits des dérivées.

4.4 Conditions de Dirichlet

Si on se contente de conditions de Neumann homogènes, l'assemblage peut en rester là. Si on souhaite des conditions de Dirichlet homogène, il y a un peu plus de travail. On doit d'abord identifier les indices des noeuds sur le bord du domaine. Nous vous fournissons une fonction qui fait cela : `boundary_nodes`, dans `Helmholtz.jl`. Pour réussir à avoir des conditions de Dirichlet homogène, il y a deux approches possibles :

1. Approche par élimination : on élimine les lignes et les colonnes correspondants aux noeuds au bord du domaine : pour chaque indice j correspondant à noeud au bord du domaine, $A_{j:} = 0$ et $A_{:j} = 0$. Après, les indices diagonaux de ces noeuds sont mis à 1 : $A_{jj} = 1$. Sur le membre de droite, on rentre les conditions de Dirichlet homogène : $\mathbf{b}_j = 0$.

- Approche par pénalisation : on pénalise les éléments diagonaux pour les obliger à être 0 (ou presque). On prend un nombre Λ très grand (par exemple $\Lambda = 10^{20}$) et pour chaque indice j correspondant à nœud au bord du domaine, $A_{jj} = A_{jj} + \Lambda$.

Tâche 5 Complétez les fonctions `apply_dirichlet_elimination` et `apply_dirichlet_penalization` dans `Helmholtz.jl`.

4.5 Résolution de l'équation

À la fin du fichier `launch_Helmholtz.jl`, vous avez un exemple de comment lancer la résolution de l'équation de Helmholtz. Pour lancer l'exemple, vous pouvez soit taper dans un terminal `julia launch_Helmholtz.jl` soit d'abord lancer `Julia` puis lancer l'instruction `include("launch_Helmholtz.jl")`.

Test 2 En partant de l'exemple `launch_Helmholtz.jl`, écrivez au moins deux tests fonctionnels permettant de tester la résolution de l'équation de Helmholtz.

5 Explication des tests

Comme vous pouvez le voir dans le répertoire, les tests peuvent être agrégés dans le répertoire `test`. Ce répertoire permet de lancer automatiquement tous les tests que vous avez rédigé ; quelques exemples vous ont été laissés pour donner une idée de comment les organiser. Pour lancer tous vos tests unitaires, suivez la procédure suivante :

- Placez vous dans le répertoire `FiniteElementSpace` et lancez `Julia` (ou alternativement, depuis `Julia`, mettez vous en mode shell avec le point virgule ; puis naviguez jusqu'au répertoire `FiniteElementSpace`).
- Activez l'environnement local : tapez le crochet fermant `]` pour passer en mode `pkg` et tapez `"activate ."` (le point est nécessaire).
- Toujours en mode `pkg`, tapez `test`. Vous lancerez alors automatiquement tous les tests listés dans `runtests.jl`, et vous obtiendrez un rapport des tests réussis ou échoués.

La fonction `test` vient aussi avec certaines options, telles que `-coverage` qui vous permet de voir la couverture de code testé.