Résolution des EDP par la méthode des éléments finis

Frédéric Hecht, Cours 5MM30 2016-2017, Master 2
Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie

8 mars 2018

Table des matières

1 Element et Différence finis en dimension 1 **4**

1.1 Notation 4

1.2 Rappels 4

1.3 Différence finis mono dimensionnel 5

1.3.1 Résolution du problème stationnaire par différence fini 5

1.3.2 Résolution du problème instationnaire par différence fini 7

1.4 Le problème modèle 10

1.5 Le problème modèle 1D stationnaire 10

1.6 Elément fini 1D 12

1.7 Construction du système linéaire 13

1.8 Méthodes directes de résolution du système linéaire 17

1.9 Formule d'intégration 18

1.10 Analyse de la méthode 19

1.11 Principe de la méthode de Galerkin 22

1.11.1 Estimation a priori 24

2 Problème Non Linéaire **26**

2.1 Rappel de Calcul différentielle 26

2.2 Des algorithmes généraux 27

2.2.1 Methode de point fixe 27

2.2.2 Methode de Newton 27

2.3 Des algorithmes de minimisation 28

2.3.1 Methode de Gradient 29

2.4 Des logiciels 29

3 Méthodes d'éléments finis P_1 Lagrange **30**

3.1 Formules de Green 30

3.2 Rappel : Forme linéaire, bilinéaire, vecteur , matrice 30

3.3 Espace affine, convexifié, et simplexe 31

3.4 Formule d'intégration 34

3.4.1 Formule d'intégration sur le triangle 34

3.5 Le maillage 36

3.6 Condition aux Limites de type Dirichlet 36

3.6.1 Méthode de pénalisation exacte 37
3.6.2 Condition de Dirichlet dans les méthodes de minimisation (GC, GMRES, ...) 39
3.7 Le problème et l'algorithme 39
3.8 La résolution d'un systéme linéaire avec le gradient conjugué 41
3.8.1 Gradient conjugué préconditionné 42
3.8.2 Test du gradient conjugué 43
3.8.3 Sortie du test 44

4 Algorithmique 47
4.1 Introduction 47
4.2 Complexité algorithmique 47
4.3 Base, tableau, couleur 49
4.3.1 Décalage d'un tableau 49
4.3.2 Renumérote un tableau 50
4.4 Construction de l'image réciproque d'une fonction 50
4.5 Construction de classe d'equivalence 51
4.6 Tri par tas (heap sort) 52
4.7 Construction des arêtes d'un maillage 53
4.8 Construction des triangles contenant un sommet donné 56
4.9 Construction de la structure d'une matrice morse 58
4.9.1 Description de la structure morse 58
4.9.2 Construction de la structure morse par coloriage 59
4.9.3 Le constructeur de la classe `SparseMatrix` 61

5 Différentiation automatique 63
5.1 Le mode direct 63
5.2 Fonctions de plusieurs variables 65
5.3 Une bibliothèque de classes pour le mode direct 66
5.4 Principe de programmation 66
5.5 Implémentation comme bibliothèque C++ 67

1 Element et Différence finis en dimension 1

1.1 Notation

d est la dimension de l'espace, $d = 1$ ou 2 dans ce cours, et Ω est un ouvert connexe de $\mathbb{R}^d$ borné.

$$x \in \mathbb{R}^d \quad \Leftrightarrow \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix}$$

Remarque, les vecteurs $\boldsymbol{u}$ seront notés généralement en police grasse, on a donc :

$$\boldsymbol{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_d \end{pmatrix}, \qquad \text{le gradient :} \qquad \nabla u = \,\mathbf{grad}\, u = \begin{pmatrix} \frac{\partial u}{\partial x_1} \\ \vdots \\ \frac{\partial u}{\partial x_d} \end{pmatrix}$$

$$\text{la divergence :} \qquad \nabla.\boldsymbol{u} = \,\mathrm{div}\,\boldsymbol{u} = \sum_{i=1}^{d} \frac{\partial u_i}{\partial x_i}$$

Le produit scalaire est noté avec un .

$$\boldsymbol{u}.\boldsymbol{v} = \sum_{i=1}^{d} u_i v_i$$

1.2 Rappels

Définition 1.1. *Dans un espace affine réel $\mathcal{A}$, le barycentre de $(\lambda_i, P_i)_{i=1,n} \in (\mathbb{R} \times A)^n$ telle que $\sum_{i=1}^{n} \lambda_i \neq 0$ est l'unique point $G = \mathrm{bar}((\lambda_i, P_i)_{i=1,n}) \in \mathcal{A}$ telle que*

$$\forall O \in \mathcal{A}, \qquad \sum_{i=1}^{n} \lambda_i \overrightarrow{OP}_i = (\sum_{i=0}^{n} \lambda_i)\overrightarrow{OG}.$$

Et si cette espace affine A est plongé dans un espace vectoriel, alors

$$G = \frac{\sum_{i=0}^{n} \lambda_i P_i}{\sum_{i=0}^{n} \lambda_i}$$

Définition 1.2. *Par définition, si f est une fonction affine de $\mathcal{A} \mapsto \mathcal{B}$, alors cette fonction commute avec les barycentres, c'est-à-dire $f(\mathrm{bar}((\lambda_i, P_i)_{i=1,n})) = \mathrm{bar}((\lambda_i, f(P_i))_{i=1,n})$, ou si l'espace affine est plongé dans un espace vectoriel, on a*

$$si \qquad \sum_{i=1}^{n} \lambda_i = 1, \qquad alors \quad f(\sum_{i=1}^{n} \lambda_i x_i) = \sum_{i=1}^{n} \lambda_i f(x_i) \tag{1}$$

De plus elle peut s'écrire comme $f(x) = A(x) + b$ où A est une application linéaire et b une constante.

- Forme bilinéaire et matrice. Soient E et F deux espaces vectoriels réels de dimension finie, soit $\{e_i/i \in I = \{1, ..., n\}\}$ une base de E, et $\{f_j/j \in J = \{1, ..., m\}\}$ une base de F, alors à toute forme bilinéaire a de $E \times F \mapsto \mathbb{R}$, on associe la matrice $A = (a_{ij})_{(i,j)\in I\times J}$ telle que $a_{ij} = a(e_i, f_j)$. Soit $l \in F'$ une forme linéaire de F.
  Et Le problème : trouver $u \in E$ tel que

$$\forall v \in F, \quad a(u, v) = l(v) \tag{2}$$

  est équivalent trouver $U \in \mathbb{R}^I$ en résolvant le système linéaire

$${}^t\!A\,U = F \quad \text{où} \quad U = (u_i)_{i\in I}, \qquad u = \sum_{i\in I} u_i e_i, \quad \text{et où} \quad F = (l(f_j))_{j\in J}.$$

- Norme des espace $L^1(\Omega)$, $L^2(\Omega)$, $L^\infty(\Omega)$

$$||u||_{L^1(\Omega)} = \int_\Omega |u|dx; \quad ||u||_{L^2(\Omega)} = (\int_\Omega |u|^2 dx)^{1/2}; \quad ||u||_{L^\infty(\Omega)} = \sup_{x\in\Omega} \text{ess}\, |u(x)| \tag{3}$$

- Produit scalaire et norme de espace $H^1(\Omega)$

$$(u, v)_{H^1(\Omega)} = \int_\Omega uv\, dx + \int_\Omega \mathbf{grad}\, u. \mathbf{grad}\, v\, dx \tag{4}$$

$$||u||_{H^1(\Omega)} = ||u||_{L^2(\Omega)} + ||(\mathbf{grad}\, u. \mathbf{grad}\, u)^{1/2}||_{L^2(\Omega)} \tag{5}$$

1.3 Différence finis mono dimensionnel

Trouver $u(x)$ fonction de $]\ell_0, \ell_1[$ tel que

$$\begin{aligned} -\frac{\partial^2 u}{\partial x^2} + au &= f \\ u(\ell_0) = g_0, \quad \frac{\partial u}{\partial x}(\ell_1) = g_1. \end{aligned} \tag{6}$$

où $\ell_0, \ell_1, g_0, g_1, a, f$ sont les données du problèmes.

1.3.1 Résolution du problème stationnaire par différence fini

Construction d'un schéma au différence fini :

Soit, M le nombre de points en espace, Notons x_i les points de calcul qui une suite strictement croissante tel que $x_0 = \ell_0$ et $x_M = \ell_1$, par exemple $x_i = \ell_0 + i\frac{\ell_1-\ell_0}{M}$. Notons $h_i = x_{i+1} - x_i$.

Soit u_i la valeur de la solution en x_i. si l'on écrit les formules de Taylor en x_i pour calculer u_{i+1} et u_{i-1}, on obtient :

$$
\begin{aligned}
u_{i+1} &= u_i + u'_i h_i \quad + \frac{1}{2} u''_i h_i^2 \quad + \frac{1}{6} u'''_i h_i^3 \quad + O(h_i^4 \quad) & (7)\\
u_{i-1} &= u_i - u'_i h_{i-1} + \frac{1}{2} u''_i h_{i-1}^2 - \frac{1}{6} u'''_i h_{i-1}^3 - O(h_{i-1}^4) & (8)
\end{aligned}
$$

Si $h_i = h$ est une constante, (7)+(8) donne :

$$
u''_i = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + O(h^2) \tag{9}
$$

et (7)-(8) donne :

$$
u'_i = \frac{u_{i+1} - u_{i-1}}{2h} + O(h^2) \tag{10}
$$

où des expression décentrées à gauche (resp.) à droite :

$$
u'_i = \frac{u_{i+1} - u_i}{h} + O(h), \quad \text{(resp.) } u'_i = \frac{u_i - u_{i-1}}{h} + O(h) \tag{11}
$$

Exercice 1. *Trouver, si h_i n'est pas constant, Construire les deux formules*

$$
u''_i = D^2_+ u_{i+1} + D^2_- u_{i-1} + D^2_. u_i + O(h^1)
$$
$$
u'_i = D^1_+ u_{i+1} + D^1_- u_{i-1} + D^1_. u_i + O(h^2)
$$

Remarque : l'on supprime les dérives troisième et on est en $O(h^3)$ où $h = max(h_i, h_{i-1})$ et l'on résout un système linéaire pour calculer les 6 coefficient $D^2_+, D^2_-, D^2_., D^1_., D^1_-, D^1_.$ en fonction de h_{i-1} et h_i.

Pour étudier la convergence de ce type de schéma, il faut montrer

Pour une bonne norme $||.||$ qui est telles que la norme du vecteur constant $v_i = 1$ $(i = 0, .., M)$ ne doit pas dépendre de M comme par exemples la norme l^∞. L'erreur $E = ||U - u^*||$ est petite (par exemple en $O(1./M)$) où le vecteurs u^* est le vecteur des valeurs de la solution exact en x_i et $U = (U_i)_{0,..,M}$ est la solution du schéma, solution du système linéaire $\mathcal{H}U = b$ avec l'opérateur linéaire $\mathcal{H}$ correspond au schéma numérique, et avec b est la partie dû au second membre et aux conditions au limites.

L'erreur de consistance du schéma est donné par $E_c = ||\mathcal{H}u^* - b||$ qui sera contrôlé avec les formules de Taylor.

L'erreur E ce calcul comme suit :

$$
\begin{aligned}
E &= ||U - u^*|| \\
&\leq ||\mathcal{H}^{-1}|| \, ||H(U - u^*)|| = ||\mathcal{H}^{-1}|| \, ||b - \mathcal{H}u^*|| = ||\mathcal{H}^{-1}|| E_c
\end{aligned}
$$

Pour que l'erreur tend vers 0 quand M tend vers ∞, il suffit que le norme matriciel de $||\mathcal{H}^{-1}||$ soit borné indépendamment de M et que l'erreur de consistance E_c tend vers 0 quand M tend vers ∞.

Il n'est pas facile généralement de montre que la norme matriciel de $||\mathcal{H}^{-1}||$ est borné indépendamment M, mais l'on peut toujours le faire numériquement avec les logiciels comme Scilal, MathLab, ...

1.3.2 Résolution du problème instationnaire par différence fini

Trouver $u(x,t)$ fonction de $]\ell_0, \ell_1[\times[0,T]$ tel que

$$\begin{aligned}\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} + au &= f \qquad (12)\\ u(.,0) = g_0, u(.,\ell_0) &= u_0, \quad \frac{\partial u}{\partial x}(.,\ell_1) = g_1.\end{aligned}$$

où $T, \ell_0, \ell_1, g_0, g_1, a, f$ sont les données du problème.

avec une discrétisation régulière en M pas en espace et N pas en temps : on cherche à approcher u en (x_i, t_j) par U_i^j avec $x_i = ih$ et $t_n = n\delta t$ et où $h = L/M$ et $\delta t = T/M$, et définisons $u_i^n = u(x_i, t_n)$.

Comme précédemment :

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_{n'}) \simeq \frac{U_{i+1}^{n'} - 2U_i^{n'} + U_{i-1}^{n'}}{h^2} \qquad (13)$$

$$\frac{\partial u}{\partial t}(x_i, t_n) \simeq \frac{U_i^n - U_i^{n-1}}{\delta t} \qquad (14)$$

Donc pour approcher le problème (18), il suffit d'utiliser (13) et (14) pour obtenir le problème approcher suivant

$$\frac{U_i^n - U_i^{n-1}}{\delta t} - \frac{U_{i+1}^{n'} - 2U_i^{n'} + U_{i-1}^{n'}}{h^2} + a_i U_i^n = f_i, \qquad (15)$$

pour $i = 1, ..., M-1$ avec les conditions aux limites suivantes $U_i^0 = u_0(ih)$, et $U_0^n = g_0$ et $\frac{U_M^n - U_{M-1}^n}{h} = g_1$.

Ici nous donne respectivement la version implicite ($n' = n$) ou explicite ($n' = n-1$) du schéma d'Euler.

Définition 1.3. *Le schéma est dit explicite quand, il ne nécessite pas de résolution de système (linéaire ou non-linéaire).*

Pour étudier la convergence de ce type de schéma, il faut montrer que l'erreur $E_n = ||U^n - u^n||$ est petite (par exemple $O(\delta t + h)$ où $U^n = (U_i^n)_{i=0,\dots,M}$ et $u^n = (u_i^n)_{i=0,\dots,M}$ pour une bonne norme.

Pour cela nous sommes amené à étudier l'opérateur linéaire $\mathcal{H}$ qui correspond au schéma numérique, tel que l'on ait $U^n = \mathcal{H}(U^{n-1}) + b^n$ où b^n est la partie constante dû à f et aux conditions au limites.

L'erreur E_n ce décompose naturellement de deux erreurs :

$$\begin{aligned}E_n &= ||U^n - u^n|| \\ &= ||\overbrace{\mathcal{H}(U^{n-1}) + b^n}^{U^n} - u^n + \overbrace{\mathcal{H}u^{n-1} - \mathcal{H}u^{n-1}}^{0}|| = ||\overbrace{\mathcal{H}U^{n-1} - \mathcal{H}u^{n-1}}^{\text{Stabilite}} + \overbrace{\mathcal{H}u^{n-1} + b^n - u^n}^{\text{Consistance}}|| \\ &\leq ||\mathcal{H}U^{n-1} - \mathcal{H}u^{n-1}|| + ||\mathcal{H}u^{n-1} + b^n - u^n||\end{aligned}$$

Définition 1.4. *L'erreur de consistance est* $||\mathcal{H}u^{n-1} - u^n + b^n||$ *et l' erreur de stabilité est* $||\mathcal{H}U^{n-1} - \mathcal{H}u^{n-1}||$.

Définition 1.5. *Un schéma sera dit consistant à l'ordre* p *en espace et* q *en temps si*

$$||\mathcal{H}u^{n-1} - u^n + b^n|| = O(h^p) + O(\delta t^q)$$

Définition 1.6. *Un schéma sera dit stable au sens de Lax et Richtmeyer si* $||\mathcal{H}^n||$ *reste borné* ($\mathcal{H}^n = \mathcal{H}\mathcal{H}^{n-1}$)*, c'est à dire*

$$\forall n < \frac{T}{\delta t}, \quad ||\mathcal{H}^n|| \leq C_s$$

Théorème 1.7. *Si un Schéma est consistant et stable au sens de Lax et Richtmeyer alors le schéma est convergent, si la solution* u *du problème continue est suffisamment régulière.*

Démonstration en exercice

Etude stabilité au sens de Von Neumann

Dans cette étude nous allons étudier le schéma via une transformée de Fourier discrète. Il faut faire les hypothèses suivantes : des conditions aux limites seront périodiques, un maillage régulier $x_j = \ell_0 + hj$ avec $h = (\ell_1 - \ell_0)/M$ en M segments, et les coefficients seront constant et le second membre nulle.

Nous allons recherche des solutions du schéma sont la forme

$$U_j^n = \sum_{k=0}^{M-1} c_k^n e^{i\pi kj/M}, \text{ avec } \qquad c_k^n = \sum_{j=0}^{M-1} \frac{U_j^n}{M} e^{i\pi - kj/M},$$

où k est la fréquence, et est le paramètre de Fourier spatial

La stabilité de Von Neumann se réduit simplement à verifier l'inégalité $c_k^n \leq c_k^{n-1}$, car les vecteurs $\omega^k = (\omega_j^k)_{j=O}^{M-1}$ définie par $\omega_j^k = e^{i\pi kj/M}$ forme une base pour $k = 0, M-1$ de $\mathbb{R}^M$ et de plus ces vecteurs sont orthogonaux pour le produit scalaire Hermitien de $\mathbb{R}^M$, on a :

$$(\omega^k, \overline{\omega^{k'}}) = \sum_{j=0}^{M-1} \omega_j^k \overline{\omega_j^{k'}} = \frac{\delta_{kk'}}{M}$$

où $\delta_{kk'}$ est le symbole de Kronecker (si $k = k'$ alors $\delta_{kk'} = 1$ sinon $\delta_{kk'} = 0$).

Donc l'étude des schémas d'Euler (15) page 7 donne donc :

$$\frac{U_j^n - U_j^{n-1}}{\delta t} - \frac{U_{j+1}^{n'} - 2U_j^{n'} + U_{j-1}^{n'}}{h^2} + aU_j^n = 0.$$

Réécrit dans la dans la base de Fourier avec le terme en n est mis a gauche de l'égalité , on a :

$$c_k^n e^{i\pi kj/M} = c_k^{n-1} e^{i\pi kj/M} + \frac{c_k^{n'} \delta t}{h^2} (e^{i\pi k(j-1)h} - 2e^{i\pi kj/M} + e^{i\pi k(j+1)h}) - \delta tac_k^n e^{i\pi kj/M}$$

après simplification par $e^{i\pi kj/M}$, il reste

$$c_k^n(1+a\delta t) = c_k^{n-1} + c_k^{n'}\frac{\delta t}{h^2}(e^{i\pi k/M} - 2 + e^{-i\pi k/M})) \tag{16}$$

comme $\cos(x) = \frac{1}{2}(e^{ix}+e^{-ix})$ et que $\cos 2x = \cos^2 x - sin^2 x = 1 - 2sin^2 x$, on a :

$$c_k^n(1+a\delta t) = c_k^{n-1} + c_k^{n'}\frac{2\delta t}{h^2}(cos(\pi k/M) - 1)) = c_k^{n-1} - c_k^{n'}\frac{4\delta t}{h^2}sin^2(\pi k/M/2) \tag{17}$$

Ce schéma d'Euler implicite ($n' = n$) est inconditionnellement stable pour $a = 0$ au sens de Von Neumann, car il suffit de remarquer que (17) implique que

$$c_k^n \leq \frac{1}{1 + a\delta t - \frac{4\delta t}{h^2}sin^2(\pi k/M/2)}c_k^{n-1}$$

et que $(1 - \frac{4\delta t}{h^2}sin^2(\pi k/M/2)) < 1$.

Etude de la stabilité au sens de Von Neumann dans le cas du schéma d'Euler explicite ($n' = n-1$).

$$c_k^n(1+a\delta t) = c_k^n(1 - \frac{4\delta t}{h^2}sin^2(\pi k/M/2))$$

il est stable si

$$-1 \leq \frac{1 - \frac{4\delta t}{h^2}sin^2(\pi k/M/2)}{(1+a\delta t)} \leq 1$$

comme l'inégalité droite est toujours vérifié, il suffit d'avoir $1 + a\delta t \geq -1 + \frac{4\delta t}{h^2}sin^2(\pi k/M/2))$ donc finalement ce schéma d'Euler explicite est stable si $\delta t \leq \frac{(2+(a\delta t))h^2}{4}$.

Exercice 2. *Faire la même étude avec l'équation convection diffusion suivante : trouver $u(x,t)$ fonction de $]\ell_0, \ell_1[\times[0,T]$ tel que*

$$\begin{aligned} \frac{\partial u}{\partial t} - \kappa\frac{\partial^2 u}{\partial x^2} + b\frac{\partial u}{\partial x} &= 0 \\ u(.,0) = u^0, \quad , u(\ell_0,.) &= u(\ell_1,.) \end{aligned} \tag{18}$$

où $T, \ell_0, \ell_1, \kappa > 0, a, b, f$ sont les données du problèmes réelles constantes et u^0 est la fonction donnée initial.

où le $\frac{\partial u}{\partial x}$ sera approché par l'une des formules (10) où (11). Parmi les six schémas implicite ou explicite, décrire lesquels sont stables en fonction du signe de b.

Etude pour pour les schémas décentre à gauche

$$\frac{U_j^n - U_j^{n-1}}{\delta t} - \kappa\frac{U_{j+1}^{n'} - 2U_j^{n'} + U_{j-1}^{n'}}{h^2} + b\frac{U_j^{n''} - U_{j-1}^{n''}}{h} = 0$$

On a après simplification par $e^{i\pi kjh}$, et en faisant les mêmes calculs avec n' , n'' prenant les valeurs n ou $n-1$ suivant les cas.

$$c_k^n = c_k^{n-1} - \kappa c_k^{n'}\frac{4\delta t}{h^2}sin^2(\pi k/M/2) - c_k^{n''}\frac{b\delta t}{h}(1 - e^{-i\pi k/M})$$

Par exemple pour le cas implicite / explicite , c'est-à-dire $n' = n$ et $n'' = n-1$ on a

$$\frac{c_k^n}{c_k^{n-1}} = \frac{1 - \frac{b\delta t}{h}(1 - e^{-i\pi k/M})}{1 + \frac{4\delta t}{h^2} sin^2(\pi k/M/2)} = \frac{1 - \frac{b\delta t}{h} + \frac{b\delta t}{h} e^{-i\pi k/M}}{1 + \frac{4\delta t}{h^2} sin^2(\pi k/M/2)}$$

pour que le module de $|\frac{c_k^n}{c_k^{n-1}}| \leq 1$ il suffit que

$$0 \leq b\delta t/h \leq 1, \tag{19}$$

cette condition (19) est appelé la condition CFL de Courant-Friedrick-Levy (1928).

Preuve : le numérateur décrit le cercle dans le plan complexe de centre $1 - \frac{b\delta t}{h}$ et de rayon $\frac{b\delta t}{h}$ qui est clairement inclus dans le cercle unité si $b > 0$. Ceci implique que le module du numérateur est plus petit ou égal à 1 et comme le dénominateur est plus grand où égal à 1, on a fini.

De plus,

— cette condition est optimale à la limite quand $M \to \infty$, Il suffit de prendre le dernier mode $k = M - 1$ et de remarquer : $\lim_{M\to\infty} e^{-i\pi(M-1)/M} = -1$;

— et si $b < 0$ alors le schéma est inconditionnellement instable , il suffit de prendre $k = 0$ et de remarquer que $\frac{c_0^n}{c_0^{n-1}} = 1 - 2\frac{b\delta t}{h} > 1$.

Donc si b est négatif, il faut faire un décentrage à droite et l'on peut donc écrire le schéma comme suit

$$\frac{U_j^n - U_j^{n-1}}{\delta t} - \kappa \frac{U_{j+1}^n - 2U_j^{n-1} + U_{j-1}^n}{h^2} + b^+ \frac{U_j^{n-1} - U_{j-1}^{n-1}}{h} - b^- \frac{U_{j+1}^{n-1} - U_j^{n-1}}{h} = 0$$

où par définition $b = b^+ - b^-$ avec $b^+ = max(b, 0)$, et $b^- = -max(-b, 0)$.

1.4 Le problème modèle

Trouver $u(t, x)$ une fonction de l'ouvert $[0, T] \times \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ telle que

$$\frac{\partial u}{\partial t} - \operatorname{div}(\, a \ \mathbf{grad}\,(u)) + (\boldsymbol{b}.\,\mathbf{grad}\, u) + c\, u = f, \quad \text{dans} \quad \Omega \tag{20}$$

$$u = u_d, \quad \text{sur} \quad \Gamma_d \tag{21}$$

$$(a\, \mathbf{grad}\, u).\boldsymbol{n} + \alpha_r u = \beta_r, \quad \text{sur} \quad \Gamma_r$$

Où $u(0, x) = u_0(x)$ est donné au temps $t = 0$, et où a,$\boldsymbol{b}$, c, u_0, u_d, α_r, β_r sont des fonctions données qui peuvent dépendre de t et de x suivante la modélisation. Et où $\boldsymbol{n}$ est le vecteur normal au bord de Ω, dirigé vers l'extérieur du domaine.

La fonction a peut même être un champ de matrices $d \times d$.

1.5 Le problème modèle 1D stationnaire

Trouver u une fonction de l'ouvert $\Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ tel que

$$-\operatorname{div}(\, a \ \mathbf{grad}\,(u)) + (\boldsymbol{b}.\,\mathbf{grad}\, u) + c\, u = f, \quad \text{dans} \quad \Omega \tag{22}$$

$$u = u_d, \quad \text{sur} \quad \Gamma_d \tag{23}$$

$$a\, \mathbf{grad}\, u.\boldsymbol{n} + \alpha_r u = \beta_r, \quad \text{sur} \quad \Gamma_r \tag{24}$$

Les données sont :

— $d = 1$, l'ouvert $\Omega =]\ell_0, \ell_1[$, avec $\ell_0 < \ell_1$ deux nombres réels donnés, avec $\Gamma_d = \{\ell_0\}$ et $\Gamma_r = \{\ell_1\}$.

— a,$\boldsymbol{b}$, c, sont des fonctions de x connues , et les termes u_d, α_r, β_r sont des constantes dans ce cas.

$\boldsymbol{n}$ est le vecteur normal exterieur au bord de Ω, c'est-à-dire que $\boldsymbol{n} = -1$ en ℓ_0 et $\boldsymbol{n} = +1$ en ℓ_1.

La formulation variationnelle est obtenue en multipliant (22) par une fonction régulière v et en intégrant.

$$-\int_\Omega \operatorname{div} (\ a \ \mathbf{grad}\,(u))v + (\boldsymbol{b}.\,\mathbf{grad}\,u)v + c\ uv\ dx = \int_\Omega fv\ dx, \tag{25}$$

On utilise la formule d'intégration par partie suivante

$$-\int_\Omega (\ a\ (u)')'vdx = \int_\Omega a\ u'v'dx - [a\ u'v]_{\ell_0}^{\ell_1}. \tag{26}$$

Comme u est connue en ℓ_0, on va prendre $v = 0$ en ℓ_0. Le terme $[a\ \ u'v]_{\ell_0}^{\ell_1}$ devient donc $a\ u'(\ell_1)v(\ell_1)$, qui après utilisation de la condition de Robin (ou Fourier suivant les auteurs) (24) et avec $\boldsymbol{n} = 1$ en ℓ_1, on obtient $a\ u'(\ell_1) = \beta_r - \alpha_r u(\ell_1)$.

Il faut dériver v et u, donc introduisons naturellement l'espace $X = H^1(\Omega)$ des fonctions de $L^2(\Omega)$ à dérivée au sens des distributions dans $L^2(\Omega)$ et l'espace X_0 l'espace des fonctions test v nulles sur Γ_d, c'est-à-dire

$$X_0 = \{v \in X/v(\ell_0) = 0\}. \tag{27}$$

Le problème se réécrit donc

Trouver $u \in X_{u_d} = \{v \in X/v(\ell_0) = u_d\}$ tel que $\forall v \in X_0$ on a

$$\int_\Omega a\ \ \mathbf{grad}\,u.\,\mathbf{grad}\,v + (\boldsymbol{b}.\,\mathbf{grad}\,u)v + c\ uv\,dx + \alpha_r u(\ell_1)v(\ell_1) = \int_\Omega fv\,dx + \beta_r v(\ell_1) \tag{28}$$

Notons par $\boldsymbol{a}$ la forme bilinéaire

$$\boldsymbol{a}(u,v) = \int_\Omega a\ \ \mathbf{grad}\,u.\,\mathbf{grad}\,v + (\boldsymbol{b}.\,\mathbf{grad}\,u)v + c\ uv\ dx\ \ + \alpha_r u(\ell_1)v(\ell_1), \tag{29}$$

et par $\boldsymbol{l}$ la forme linéaire

$$\boldsymbol{l}(v) = \int_\Omega fv\ dx\ \ + \beta_r v(\ell_1). \tag{30}$$

Le problème ce réduit formellement à

Trouver $u \in X_{u_d}$ tel que

$$\forall v \in X_0; \qquad \boldsymbol{a}(u,v) = \boldsymbol{l}(v). \tag{31}$$

Ce problème n'est pas linéaire, il est affine. Pour le rendre linéaire, il suffit de choisir une fonction $\tilde{u}_d$ qui vérifie la condition de Dirichlet (c'est-à-dire $\tilde{u}(x) = u_d(x)$ pour $x \in \Gamma_d$).

Notons $\tilde{u} = u - \tilde{u}_d$, on a $\tilde{u} \in X_0$ et $u = \tilde{u} + \tilde{u}_d$. Le problème précédent est équivalent au problème linéaire suivant :

Trouver, $\tilde{u} \in X_0$ tel que

$$\forall v \in X_0; \qquad \boldsymbol{a}(\tilde{u}, v) = \boldsymbol{l}(v) - \boldsymbol{a}(\tilde{u}_d, v) \tag{32}$$

Le système linéaire associé est clairement carré, les inconnues et les fonctions test sont dans le même espace X_0.

Si l'espace X_0 était de dimension finie, on saurait écrire le système linéaire associé. Mais bien sûr X_0 n'est pas de dimension finie et on va approcher X_0 par une famille d'espaces X_{0h} de dimension finie. Le paramètre h est fait pour tendre vers 0. Cette famille d'espaces X_h de dimension finie dans X vont approcher X. C'est à dire que, par exemple, on a l'estimation suivante

$$\inf_{v_h \in X_h} ||u - v_h||_X \leq Ch^s ||v||,$$

où C et s sont deux constantes strictement positives données. Notons $X_{h0} = X_h \cap X_0$.

Alors, on peut résoudre le problème approché dans X_{h0} :

Trouver $\tilde{u}_h \in X_{0h}$ tel que :

$$\forall v_h \in X_{h0}; \qquad \boldsymbol{a}(\tilde{u}_h, v_h) = \boldsymbol{l}(v_h) - \boldsymbol{a}(\tilde{u}_{0h}, v_h) \tag{33}$$

où $\tilde{u}_{0h}$ est une fonction de X_h qui relève les conditions aux limites de Dirichlet, c'est-à-dire telle que $\tilde{u}_{0h}(\ell_0) = u_d$.

1.6 Elément fini 1D

Soit $q^i, i = 0, ..., N$ une suite strictement croissante de $N+1$ réels telle que $q^0 = \ell_0$ et $q^N = \ell_1$. Nous noterons par K_k l'intervalle $]q^{k-1}, q^k[$, et $|K_k| = q^k - q^{k-1}$ la mesure de K_k. Nous appelons $h = \sup_{k=1}^N |K_k|$, et par abus de langage nous noterons $\mathcal{T}_{d,h} = \{K_k, k = 1, ..., N\}$ un maillage de Ω de taille maximale h. Les intervalles K_k seront appelés les élements de $\mathcal{T}_{d,h}$. Soit K un élément de $\mathcal{T}_{d,h}$, nous notons i_0^K, i_1^K les deux sommets de K tels que $i_0^K < i_1^K$ et notons k^K le numéro de l'élément K, c'est à dire :

$$K =]q^{i_0^K}, q^{i_1^K}[, \qquad k^K - 1 = i_0^K \qquad k^K = i_1^K, \qquad k = k^{K_k}.$$

Associé à ce maillage mono-dimensionnel, nous définissons l'espace des éléments finis P_1 Lagrange comme suit :

$$X_h = \{v \in \mathcal{C}^0(\Omega) / \; \forall K \in \mathcal{T}_{d,h}, \quad v_{|K} \in P_1(K)\} \quad \subset H^1(\Omega) \tag{34}$$

— $v_{|K}$ est la fonction restriction de v sur l'intervalle K qui est définie par

$$v_{|K} : \quad x \in K \mapsto v(x).$$

— $P_1(K)$ est l'espace des fonctions polynômes de degré inférieur ou égal à 1 de l'intervalle K. C'est-à-dire que $v_{|K}(x) = a_K x + b_K$.

On remarque que les fonctions de $H^1(\Omega)$ sont des fonctions continues sur Ω dans le cas mono-dimensionnel, ce qui est malheureusement faux si $d > 1$. Et les fonctions de $\mathcal{C}^0(\Omega)$ à dérivée bornée sont toujours dans $H^1(\Omega)$ si l'ouvert Ω est borné. C'est pour cette raison que l'on a une approximation interne.

Lemme 1.8. *Les fonctions de X_h sont uniquement définies par leurs valeurs aux points q^i, pour $i = 0, ..., N$.*

Effectivement, notons le deux fonctions de base λ_0^K et λ_1^K de $P_1(K_k)$ suivante :

$$\lambda_0^K(x) = \frac{x - q^{i_1^K}}{q^{i_0^K} - q^{i_1^K}}, \qquad \text{et} \qquad \lambda_1^K(x) = \frac{q^{i_0^K} - x}{q^{i_0^K} - q^{i_1^K}} \tag{35}$$

Remarquons que $\lambda_0^K(q^{i_0^K}) = 1$ et $\lambda_0^K(q^{i_1^K}) = 0$ et inversement $\lambda_1^K(q^{i_0^K}) = 0$ et $\lambda_0^K(q^{i_1^K}) = 1$, ce que l'on peut résumer par $\lambda_l^K(q^{i_m^K}) = \delta_{lm}$ pour $l, m = 0$ ou 1 (où δ_{lm} est le symbole de Kroneker). Ces 2 fonctions sont appelées les coordonnées barycentrique de K, car on a les deux égalités triviales fondamentales suivantes

$$x = \lambda_0^K(x) q^{i_0^K} + \lambda_1^K(x) q^{i_0^K} \quad \text{et} \quad \lambda_0^K(x) + \lambda_1^K(x) = 1 \tag{36}$$

Donc, sur tout intervalle K de $\mathcal{T}_{d,h}$, on a

$$\forall v_h \in X_h; \quad v_{h|K}(x) = v(q^{i_0^K}) \lambda_0^K(x) + v(q^{i_1^K}) \lambda_1^K(x) \tag{37}$$

Définition 1.9. *Notons $\{w_i, i = 0, ..., N\}$ l'ensemble des fonctions de X_h telles que*

$$w_i(q^j) = \delta_{ij}.$$

Alors cet ensemble est une base de X_h, et l'ensemble $\{w_i, i = 1, ..., N\}$ est une base de $X_{0h} = X_h \cap X_0 = \{v_h \in X_h / v_h(\ell_0) = 0\}$. De plus on a

$$\forall v \in X_h \qquad v = \sum_{i=0}^{N} v(q^i)\, w_i \tag{38}$$

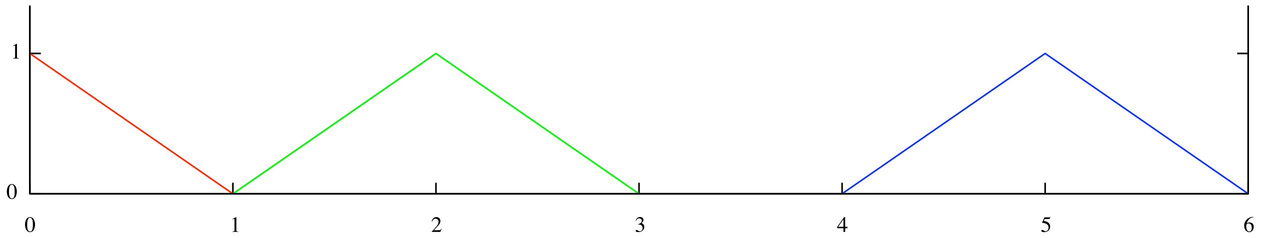

FIGURE 1 – Les trois fonctions de base respectivement w_0, w_2 et w_5.

1.7 Construction du système linéaire

Pour simplifier, toutes le données sont supposées dans un premier temps constantes, ce sont donc des nombres réels.

Lemme 1.10. *Le système linéaire $N \times N$ à résoudre $A_0 U_0 = F_0$ est défini par*

$$A_0 = (a_{ij})_{i,j=1,\dots,N} \quad \text{avec} \quad a_{ij} = \boldsymbol{a}(w_j, w_i)$$

Attention à la transposition des indices i **et** j, *car les lignes correspondent au fonction test v par construction.*

et

$$F_0 = (f_i)_{i=1,..,N} \quad \text{avec} \quad f_i = \boldsymbol{l}(w_i) - a(\tilde{u}_{dh}, w_i)$$

et où l'on a défini $\tilde{u}_{dh} = u_d\, w_0$.

Pour finir la solution du problème est la fonction suivante

$$u = \sum_{j=0}^{N} u_j\, w_j \qquad \text{avec} \quad u_0 = u_d \quad \text{et avec} \quad U_0 = (u_i)_{i=1}^{N}$$

Ce système peut être réécrit plus simplement pour se rapprocher de l'informatique comme le système $(N+1) \times (N+1)$ suivant :

$$AU = F, \quad A = (a_{ij})_{i,j=0,\dots,N}, \quad F = (f_i)_{i=0}^{N}, \quad U = (u_i)_{i=0}^{N} \tag{39}$$

où $a_{0,j} = \delta_{0j}$, $a_{ij} = \boldsymbol{a}(w_j, w_i)$, si $i > 0$, $f_0 = u_d$, et $f_i = \boldsymbol{l}(w_i)$ si $i > 0$,.

La construction de la matrice A et du second membre F :

Notons $\boldsymbol{a}_K$ la forme bilinéaire suivante :

$$\boldsymbol{a}_K(u,v) = \int_K a\ \mathbf{grad}\, u.\, \mathbf{grad}\, v + (\boldsymbol{b}.\, \mathbf{grad}\, u)v + c\, uvdx, \tag{40}$$

et $\boldsymbol{l}_K$ la forme linéaire

$$\boldsymbol{l}_K(v) = \int_K f v\, dx. \tag{41}$$

On a clairement

$$\boldsymbol{a}(u,v) = \sum_{K \in \mathcal{T}_{d,h}} \boldsymbol{a}_K(u,v) + \alpha_r u(\ell_1) v(\ell_1)$$

et

$$\boldsymbol{l}(v) = \sum_{K \in \mathcal{T}_{d,h}} \boldsymbol{l}_K(u,v) + \beta_r v(\ell_1)$$

Remarque 1. $\boldsymbol{a}_K(w_i, w_j) = 0$ *si i, j ne sont pas des sommets $\{i_0^K, i_1^K\}$ de K alors les seuls éléments non nuls sont pour (i,j) dans $\{i_0^K, i_1^K\}^2$, soit quatre valeurs. Donc, la matrice A_K associé à $\boldsymbol{a}_K$ est une matrice de la forme*

$$A_K = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \breve{a}_{00}^K & \breve{a}_{01}^K & 0 & 0 \\ 0 & 0 & \breve{a}_{10}^K & \breve{a}_{11}^K & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{42}$$

pour l'élément K_3.

$$\breve{a}^K_{lm} = \int_K a \ \mathbf{grad}\,\lambda^K_m . \mathbf{grad}\,\lambda^K_l + (\boldsymbol{b}. \mathbf{grad}\,\lambda^K_m)\lambda^K_l + c\lambda^K_m\lambda^K_l dx$$

ce terme $\breve{a}^K_{lm}$ se place en $a_{i^K_l i^K_m}$ avec i^K_0, i^K_1 les deux numéros des sommets de K, c'est-à-dire que $K =]q^{i^K_0}, q^{i^K_1}[$.

Lemme 1.11. *La matrice A est tridiagonale.*

Calcul de la matrice $\breve{A}^K$

Soit K l'élement K de sommets i^K_0, i^K_1, avec $i^K_0 < i^K_1$, notons $h_K = (q^{i^K_1} - q^{i^K_0})$, on a

- $\mathbf{grad}\,\lambda^K_m = -1^{\delta_{0m}}/h_K$
- $\int_K \lambda^K_m dx = h_K/2$
- $\int_K \lambda^K_m \lambda^K_l dx = h_K(1 + \delta_{lm})/6$

Donc si a, $\boldsymbol{b}$, c sont des fonctions constantes, la matrice $\breve{A}^K$ (2×2) est définie par :

$$\forall (l, m) \in \{0, 1\}^2, \qquad \breve{a}^K_{lm} = (-1)^{1+\delta_{lm}} \frac{a}{h_k} + (-1)^{\delta_{m0}} \frac{\boldsymbol{b}}{2} + h_K c \frac{1 + \delta_{lm}}{6}$$

Exercice 3. Ecrire une fonction C++ qui ajoute à une matrice tri-diagonale A, la matrice A^K, dans le cas où les fonctions a, $\boldsymbol{b}$, c, f sont constante.
Puis, écrire une fonction qui construit la matrice tri-diagonale A complète définie en (39), et une autre fonction pour construire le seconde membre associé F.
La matrice tri-diagonale pourra être modélisée avec la classe C++ suivante :

```
class MatTriDia { public:
  int n;
  vector<double> a,b,c;
  MatTriDia(int nn) : n(nn),a(n),b(n),c(n) {MiseAZero();}
  void MiseAZero() { for (inti=0;i<n;++i) a[i]=b[i]=c[i]=0.;}
  double & operator()(int i,int j) {
     if(i==j-1) return a[i] ;
     else if (i==j) return b[i];
     else if (i==j+1) return c[i];
     else {cerr << "..\n" ; abort();} } //    #include <cstdlib>
 };
```

Pour un objet défini par `MatTriDia A(n);` on aura les correspondances suivantes :
`A.a(i)` $\equiv a_{i,i-1}$, `A.b(i)` $\equiv a_{i,i}$, `A.c(i)` $\equiv a_{i,i+1}$.
La matrice A de taille n est donc de la forme :

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & . & . & 0 & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & . & a_{n-2} & b_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & . & . & . & a_{n-1} & b_{n-1} \end{pmatrix} \tag{43}$$

Attention à la correspondance entre N et n.

Exercice 4. Voila une classe `MatElement` qui représente une matrice A_K.

```
class MatElement { public:
  int iK[2];       //    les indices où l'on met la matrice 2x2
  double  a[2][2];                          //    les 2x2 coef
  MatElement(int i1,int i2,double a00,double a01,
                          double a10,double a11)
   {
     ii[0]=i1; ii[1]=i2;
     a[0][0]=a00;
     a[1][0]=a10;
     a[0][1]=a01;
     a[1][1]=a11;
  };
  };
```

1. Ajouter à la classe `MatTriDia`, l'opérateur
   `MatTriDia & MatTriDia::operator+= (MatElem &Ak);`
2. Construire une fonction `AK` qui retourne la matrice élémentaire A_K de prototype :
   ```
   MatElement AK(double qk1,double qk,int k,double a,
                 double b, double c);
   ```

1.8 Méthodes directes de résolution du système linéaire

Exercice 5. Ajouter à la classe `MatTriDia`, l'opérateur
`MatTriDia & MatTriDia::operator+= (MatElem &Ak);`
où la classe `MatElem` représente une matrice élémentaire.

Les matrices tri-diagonales sont très faciles à factoriser par la méthode de Gauss. Si le système $AX = F$ est écrit sous la forme générale ($a_0 = c_{n-1} = 0$ par convention)

$$\begin{pmatrix} b_0 & c_0 & 0 & 0 & . & . & 0 & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & . & a_{n-2} & b_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & . & . & . & a_{n-1} & b_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ . \\ . \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ . \\ . \\ f_{n-2} \\ f_{n-1} \end{pmatrix} \tag{44}$$

la matrice A peut se décomposer sous la forme $A = LU$, avec les matrices

$$L = \begin{pmatrix} b_0^* & 0 & 0 & . & . & 0 & 0 \\ a_1 & b_1^* & 0 & 0 & . & 0 & 0 \\ . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & a_{n-2} & b_{n-2}^* & 0 \\ 0 & 0 & 0 & 0 & . & a_{n-1} & b_{n-1}^* \end{pmatrix}, U = \begin{pmatrix} 1 & c_0^* & 0 & . & . & 0 & 0 \\ 0 & 1 & c_1^* & 0 & . & 0 & 0 \\ . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & 1 & c_{n-2}^* \\ 0 & 0 & 0 & 0 & . & 0 & 1 \end{pmatrix}.$$

Exercice 6. Ajouter à la classe `MatTriDia` la méthode `void LU();` qui remplace les vecteurs b,c par leurs correspondants b^*,c^*.

En identifiant les coefficients, nous obtenons les récurrences suivantes pour le calcul des coefficients b^* et c^* :

$$\left\{\begin{array}{l} b_0^* = b_0 \\ c_0^* = c_0/b_0 \end{array}\right. \qquad \left\{\begin{array}{l} b_k^* = b_k - a_k \cdot c_{k-1}^* \\ c_k^* = c_k/b_k^* \end{array}\right. \qquad k = 1 \dots n-1. \tag{45}$$

Remarque 2. *Observons que la factorisation LU est possible si les coefficients b_k^* sont non-nuls, donc la matrice A doit être inversible.*

Le système peut être résolu maintenant en deux étapes $AX = F \Longrightarrow L\underbrace{UX}_{Y} = F$.

$$LY = f \Longrightarrow \left\{\begin{array}{l} y_0 = f_0/b_0^* \\ y_k = (f_k - a_k\, y_{k-1})\,/b_k^*, \quad k = 1 \dots n-1 \end{array}\right. \tag{46}$$

$$UX = Y \Longrightarrow \left\{\begin{array}{l} x_{n-1} = Y_{n-1} \\ x_k = y_k - c_k^*\, x_{k+1}, \quad k = (n-2) \dots 0 \end{array}\right. \tag{47}$$

Remarque 3. *Pour la programmation, nous utiliserons seulement quatre vecteurs a, b, c, f contenant initialement les coefficients a_k, b_k, c_k, f_k du système. Les récurrences (45) seront groupées dans la même boucle, ce qui permet de stocker b^* dans b et c^* dans c. Pour la boucle ascendante (46) Y peut être stocké dans F et, finalement, dans la boucle descendante (47), la solution X du système sera être stockée également dans F.*

Exercice 7. Ajouter la méthode `void Solve(vector<double> &F);` à la classe `MatTriDia` qui résout le système $AX = F$, si la matrice A est factorisée sous la forme LU. La fonction retournera la solution dans le vecteur F.

1.9 Formule d'intégration

Comme vous avez pu le remarquer, nous avons besoin de calculer des intégrales sur des segments. Voilà quelques formules pour approcher ces intégrales.

Soit $K =]q^i, q^j[$ un intervalle ($i = i_0^K$ et $j = i_1^K$), notons $h_K = |q^i - q^j|$ la longueur de l'intervalle.

Une formule d'intégration s'écrit

$$\int_K f dx \sim h_K \sum_{\ell=1}^{N} \omega_k f(\xi_\ell^K) \tag{48}$$

Les points integrations ξ_ℓ^K sont donnés dans un élément de référence $\hat{K}$ ici l'élément $\hat{K} =]0,1[$. Ces points dans l'élément de référence $\hat{K}$ sont notés $\hat{\xi}_\ell$ et on a

$$\xi_\ell^K = (1 - \hat{\xi}_\ell) q^i + \hat{\xi}_\ell q^j \tag{49}$$

De plus, on remarquera que l'on a :

$$\lambda_0^K(\xi_\ell^K) = 1 - \hat{\xi} \qquad \lambda_1^K(\xi_\ell^K) = \hat{\xi} \tag{50}$$

donc pour on a

$$w_i(\xi_\ell^K) = 1 - \hat{\xi}_\ell \quad \text{et} \quad w_j(\xi_\ell^K) = \hat{\xi}_\ell \tag{51}$$

ce qui simplifie grandement, utilisation des formule d'intégration avec les fonctions de base de l'élément K.

Les paramètres des formules sont N le nombre points d'intégrations, les points $\hat{\xi}_\ell$ et poids ω_ℓ d'intégration, l'ordre de la formule, et pour quelle type de polynôme la formule est exact.

Voilà une petite liste de formule d'intégration bien utile :

N	`ordre`	$\hat{\xi}_\ell$	ω_ℓ	exact sur $P_k, k =$
1	2	$1/2$	h_K	1
2	4	$(1 \pm \sqrt{1/3}\ )/2$	$h_K/2$	3
3	6	$(1 \pm \sqrt{3/5})/2$ $1/2$	$(5/18)h_K$ $(8/18)h_K$	5
4	8	$(1 \pm \frac{\sqrt{525+70\sqrt{30}}}{35})/2$ $(1 \pm \frac{\sqrt{525-70\sqrt{30}}}{35})/2$	$\frac{18-\sqrt{30}}{72}\ h_K$ $\frac{18+\sqrt{30}}{72} h_K$	7
5	10	$(1 \pm \frac{\sqrt{245+14\sqrt{70}}}{21})/2$ $1/2$ $(1 \pm \frac{\sqrt{245-14\sqrt{70}}}{21})/2$	$\frac{322-13\sqrt{70}}{1800} h_K$ $\frac{64}{225} h$ $\frac{322+13\sqrt{70}}{1800} h_K$	9
2	2	$1/2 \pm 1/2$	$h_K/2$	1

1.10 Analyse de la méthode

L'un point de vue théorique notre problème est de la forme suivante :

Définition 1.12 (Le problème abstrait). *Soit H un espace de Hilbert. soit $\boldsymbol{a}$ une forme bilinéaire sur H et $\boldsymbol{l}$ un forme linéaire sur H.*

Le problème formelle est : trouvez $u \in H$, telle que

$$\forall v \in H; \qquad \boldsymbol{a}(u, v) = l(v). \tag{52}$$

Définition 1.13. *Un problème est dit bien posé s'il existe une silution unique et qu'elle dépend continûment des données.*

Le théorème suivant nous donne les hypothèses pour que le problème soit bien posé.

Théorème 1.14 (Banach-Necas-Babuska)**.** *Si la forme bilinéaire* $\boldsymbol{a}$ *est continue et la forme linéaire* $\boldsymbol{l}$ *sont continue sur* H*, c'est à dire*

$$\boldsymbol{a}(u,v) \leq c_M \,||u||\,||v||, \qquad \boldsymbol{l}(v) \leq c_l ||v||$$

le problème (52) est bien pose si et seulement si la condition inf-sup suivante est vérifé

$$\inf_{u\in H} \sup_{v\in H} \boldsymbol{a}(u,v) \geq \alpha ||u||\,||v||, \qquad \alpha > 0$$

et la condition : $(\forall u \in H; \quad \boldsymbol{a}(u,v) = 0) \Rightarrow v = 0.$

Démonstration. Notons $A : u \in H \mapsto a(u,.) \in H'$, où H' est l'ensemble des formes linéaires continue de H, la norme pour $f \in H'$ est

$$||f|| = \sup_{v\in H, v\neq 0} \frac{f(v)}{||v||}.$$

Donc, pour tout $v \in H$, on a $\alpha||u|| \leq ||A(u)|| \leq c_M||u||$, donc A est injective.

Maintenant, montrons que A est sujective, pour cela démontrons que $A(H)$ est fermé. Il suffit de montrer que toute suite y_n un suite de Cauchy de $A(H)$ converge. Notons la suite x_n tel que $A(x_n) = y_n$ (A est injective), la condition inf-sup nous donne

$$||y_n - y_m|| = ||A(x_m - y_m)|| = \sup_{v\in H, v\neq 0} \frac{a(x_n - x_m, v)}{||v||} \geq \alpha ||x_n - x_m||$$

La suite x_n est de cauchy, et donc converge vers x (H est complet) et par continuité la suite de Cauchy y_n converge vers $A(x)$ donc $A(H)$ est bien fermé. La second condition nous montre que l'orthognonal de $A(H)$ est réduit à 0, comme $A(H)$ est ferme, on a $A(H) = H'$, d'où A est inversible et continue de H dans H', ce qui termine la première implication.

La réciproque est trivial, si l'on remarque qu'un problème est bien posé implique que A à un inverse continue. □

Pour utiliser ce théoreme, il suffit de construire un $v_u \in H$ telle que $a(u, v_u) \geq \alpha_1 ||u||^2$ et $||v_u|| \leq \alpha_2 ||u||$, alors la condition inf-sup est vérifie pour $\alpha = \frac{\alpha_1}{\alpha_2}$.

Définition 1.15 (Coercivité). *On dit qu'une forme bilinéaire* $\boldsymbol{a}$ *de* $H \times H$*, est coersive, si il existe une constante réelle* $c_m > 0$ *telle que*

$$\forall v \in H; \qquad c_m ||v||^2 \leq \boldsymbol{a}(v,v) \tag{53}$$

Lemme 1.16 (Lax-Milgram)**.** *Si la forme bilinéaire* $\boldsymbol{a}$ *de* $H \times H$*, est coercive et continue, et si forme linéaire* $\boldsymbol{l}$ *est continue, alors le problème (52) est bien posé.*

Démonstration. Il suffit d'utiliser le théorème précédent, en choisissant $v_u = u$ pour la condition inf-sup. Pour la second condition, il suffit de remarquer que si $\boldsymbol{a}(v, v) = 0$ alors $c_m||v||^2 \leq \boldsymbol{a}(v, v) = 0$, et donc $v = 0$. □

Lemme 1.17 (Inégalite de Poincaré)**.** *Soit Ω un ouvert connexe borné de $\mathbb{R}^d$, soit Γ^d une partie régulière de du bord de Ω de mesure positive , alors sur le sous espace X_0 des fonctions de $H^1(\Omega)$ nulle sur Γ_d, il existe une constant C telle que*

$$\forall v \in X_0; \qquad \int_\Omega u^2 dx \leq C \int_\Omega ||\,\mathbf{grad}\, u||^2 dx. \tag{54}$$

Cette inégalité, implique que la semi-norme suivante

$$|u|_{H_1(\Omega)} = \left(\int_\Omega ||\,\mathbf{grad}\, u||^2 dx \right)^{\frac{1}{2}}$$

est une norme équivalente à la norme $H^1(\Omega)$ sur l'espace X_0.

Démonstration. Premièrement, Si $X_0 = H^1_0(\Omega)$ alors, Comme $\mathcal{D}(\Omega)$ est dense dans $H^1_0(\Omega)$ il suffit d'écrire une majoration pour les fonctions de $\mathcal{D}(\Omega)$.

En intégrant sur l'axe de x on a

$$u(x, ..) = \int_{x_\Gamma(...)}^x \partial_x u(\xi, ...) d\xi, \text{ avec } x_\Gamma(...) \in \partial\Omega$$

Puis en utilisant l'inégalite de Cauchy-Schawrz on a

$$u(x, ...)^2 = (\int_{x_\Gamma(...)}^x \partial_x u(\xi, ...) d\xi)^2 \leq (\int_{x_\Gamma(...)}^x (\partial_x u(\xi, ...))^2 d\xi) \int_{x_\Gamma(...)}^x 1 d\xi$$

Donc, on en utilisant le théorème de Fubini ;

$$||u||^2_{L^2} \leq ||\partial_x u||^2_{L^2} (\int_\Omega (x - x_\Gamma(...))^2$$

Pour finir,il suffit de remarquer que $(\int_\Omega (x - x_\Gamma(...))^2$ est borné.

Sinon, comme l'ouvert est borné et régulier et connexe , soit x_0 sur le bord de l'ouvert, pour tout point x de l'ouvert il existe une courbe γ_x C^∞ par morceaux qui va de x_0 à x de paramètre par l'abscisse curviligne de plus les longueurs des γ_x est L_x borné, par L suffisamment grand.

Il suffit de refaire la même preuve avec les paramétrisations γ_x.

□

Montrons que notre problème mono-dimensionnel est bien posé sous les hypothèses suivantes.

Théorème 1.18. *Si la mesure de Γ_d est strictement positive, si les fonctions sont telles que $a \in L^\infty(\Omega)$, $\boldsymbol{b} \in W^{1,\infty}(\Omega)$, $c \in L^\infty(\Omega)$, si il existe une constant réel c_a, telle que $0 < c_a \leq a$, $c - \frac{\boldsymbol{b}'}{2} \geq 0$, $0 \leq \alpha_r$, $\boldsymbol{b}.\boldsymbol{n} \geq 0$ sur Γ_r alors le problème de bien posé.*

Démonstration. Nous allons utiliser le lemme de Lax-Milgram.

Premièrement, il est évident $\boldsymbol{a}$ est bien continue.

On a

$$a(v,v) = \int_\Omega a||\,\mathbf{grad}\,v||^2 dx + \int_\Omega (\boldsymbol{b}.\,\mathbf{grad}\,v)v dx + \int_\Omega cv^2 dx + \underbrace{\alpha_r v(\ell_1)^2}_{\geq 0}.$$

En notant $(\boldsymbol{b}v^2)' = \boldsymbol{b}'v^2 + 2\boldsymbol{b}v'v$, on obtient

$$\begin{aligned} \int_{\ell_0}^{\ell_1} ((\boldsymbol{b}.v')v \ + \ cv^2 \ dx &= \int_{\ell_0}^{\ell_1} \frac{1}{2}(\boldsymbol{b}v^2)' + (c - \frac{1}{2}\boldsymbol{b}')v^2 \ dx \\ &= \underbrace{-(\boldsymbol{b}v^2)(\ell_0)}_{=0} + \underbrace{(\boldsymbol{b}v^2)(\ell_1)}_{\geq 0} + \int_{\ell_0}^{\ell_1} \underbrace{(c - \frac{1}{2}\boldsymbol{b}')}_{\geq 0} v^2 \ dx \quad \geq 0 \end{aligned}$$

D'où

$$a(v,v) \geq \int_\Omega a||\,\mathbf{grad}\,v||^2 dx \geq c_a \int_\Omega ||\,\mathbf{grad}\,v||^2 dx = c_a |v|_{H_1(\Omega)}.$$

Comme la mesure de Γ_d est strictement positive, l'inégalité de Poincaré, nous dit que la semi-norme $|.|_{H_1(\Omega)}$ équivalente u sur X_0.

□

Maintenant nous allons faire l'analyse de l'erreur.

1.11 Principe de la méthode de Galerkin

Soit H_h un sous espace de dimension fini de H,

Définition 1.19. *Notre problème discret est : trouvez $u_h \in H_h$, telle que*

$$\forall v_h \in H_h; \qquad \boldsymbol{a}(u_h, v_h) = l(v_h). \tag{55}$$

Lemme 1.20 (de Cea)**.** *Soit une la forme bilinxéaire $\boldsymbol{a}$ de $H \times H$ (H espace de Hilbert) coercive et continue, c'est a dire qu'il existe deux contantes réels strictement positives c_m et c_M telles que*

$$\forall (v,w) \in H^2, \qquad ||v||^2 c_m \leq \boldsymbol{a}(v,v) \quad et \quad \boldsymbol{a}(v,w) \leq c_M ||v||\,||w||$$

et si forme linéaire $\boldsymbol{l}$ est continue, alors nous avons l'estimation d'erreur entre u solution de (52) et u_h solution de (55) suivante :

$$\forall v_h \in H_h, \qquad ||u - u_h||_H \leq \frac{c_M}{c_m} ||u - v_h||_H \tag{56}$$

Démonstration. Remarquons la relation orthogonalité suivant

$$\forall w_h \in H_h; \qquad a(u - u_h, w_h) = l(v_h) - l(v_h) = 0 \tag{57}$$

On a pour tout v_h dans H_h

$$\begin{aligned} c_m ||u - u_h||_H^2 \le a(u - u_h, u - u_h) &= a(u - u_h, u - u_h + \underbrace{u_h - v_h}_{w_h \in H_h}) \\ &= a(u - u_h, u - v_h) \\ &\le c_M ||u - u_h||_H \, ||u - v_h||_H \end{aligned}$$

En divisant par $c_m ||u - u_h||_H$, on obtient le résultat. □

Ce résultat est fondamental, car il montre que si u est approchable à ε près dans H_h, alors $||u - u_h||$ est de l'ordre de ε.

Dans le cas plus général où on a juste une condition inf-sup, il faut utiliser le premier lemme de Strang que voici :

Lemme 1.21 (Strang)**.** *Sous les mêmes hypothèse que le théorème Banach-Necas-Babuska 1.14 et sous l'Hypothèse de la condition inf-sub discrète :*

$$\inf_{u_h \in H_h} \sup_{v_h \in H_h} \frac{\boldsymbol{a}(u_h, v_h)}{||u_h||_H \, ||v_h||_H} \ge \alpha^* > 0$$

indépendant du paramètre de discrétisation h.

alors on a

$$||u - u_h||_H \le (1 + \frac{C_M}{\alpha^*}) ||u - v_h||_H$$

Démonstration. Notre condition inf-sup discret :

$$\alpha^* ||u_h - v_h||_H \le \sup_{w_h \in H_h} \frac{\boldsymbol{a}(u_h - v_h, w_h)}{||w_h||}$$

et La relation d'orthogonalité (57) nous donne

$$\boldsymbol{a}(u_h - v_h, w_h) = \boldsymbol{a}(u_h - u + u - v_h, w_h) = \boldsymbol{a}(u - v_h, w_h)$$

implique

$$\alpha^* ||u_h - v_h||_H \le \sup_{w_h \in H_h} \frac{\boldsymbol{a}(u - v_h, w_h)}{||w_h||} \le C_M ||u - v_h||_H$$

Et donc l'inégalité précédente , plus inégalité triangulaire donne :

$$||u - u_h||_H \le ||u_h - v_h||_H + ||v_h - u||_H \le (1 + \frac{C_M}{\alpha^*}) ||v_h - u||_H$$

□

1.11.1 Estimation a priori

Si la solution est continue, et si on utilise la méthode des éléments finis.

Notons $\mathcal{I}_h(v)$ l'interpolé de v, la fonction de H_h telle que $\mathcal{I}_h(v)(q) = v(q)$ aux $N+1$ points q du maillage, c'est à dire que

$$\mathcal{I}_h(v) = \sum_{i=0}^{N} v(q^i) w_i \tag{58}$$

où les w_i sont les fonctions de base de H_h de la définition 1.9.

Maintenant, nous allons estimé $||u - \mathcal{I}_h u||_{H^1(\Omega)}$, sur un élément $K =]a, b[$, de longueur $h_K = b - a$ (*Attention, a et b non rien avoir avec les coefficients du problème original*).

Notons $W^{2,\infty}(K)$, espace de Sobolev : l'ensemble des fonctions réelles de l'intervalle K à dérivées première et secondes bornées presque partout (ces fonctions sont continues).

Lemme 1.22. *Soit $u, f, g \in W^{2,\infty}(]a,b[)^3$ telles que $f(a) \leq u(a) \leq g(a)$, $f(b) \leq u(b) \leq g(b)$ et telles que $f'' \geq u'' \geq g''$, alors nous avons $f \leq u \leq g$.*

Démonstration. La preuve est basée sur la remarque qu'une fonction convexe sur $[a, b]$ et négative en a et b est négative sur $[a, b]$ entier. Pour finir, utilisons cette remarque avec $f - u$ et $u - g$. □

Soit $[a, b]$ un intervalle de $\mathbb{R}$, et soit $u \in W^{2,\infty}(]a, b[)$ une fonction continue, soit $\mathcal{I}_1(u)$ l'interpolation P_1 de u , l'interpolé est défini par $\mathcal{I}_1(u) : t \mapsto (1-t)u(a) + tu(b)$.

Pour obtenir les majorations d'erreurs classiques, il suffit utiliser comme fonction majorante (resp. minorante) la fonction $f(x) = \frac{1}{2}\inf_K(u'')(x-a)(x-b)$ (resp. $g(x) = \frac{1}{2}\sup_K(u'')(x-a)(x-b)$). Le lemme 1.22 nous donne l'encadrement suivant :

$$\frac{1}{2}\sup_K(u'')(x-a)(x-b) \quad \leq \quad (u - \mathcal{I}_1 u) \quad \leq \quad \frac{1}{2}\inf_K(u'')(x-a)(x-b) \tag{59}$$

si $u \in W^{2,\infty}(]a, b[)$.

On en déduit, l' erreur d'interpolations P_1 dans trois normes classique sur l'élément K :

$$\|u - \mathcal{I}_1 u\|_{L^\infty} \leq \frac{h_K^2}{8} ||u''||_\infty \tag{60}$$

$$\|u - \mathcal{I}_1 u\|_{L^1} \leq \frac{h_K^3}{12} ||u''||_\infty \tag{61}$$

$$\|u - \mathcal{I}_1 u\|_{L^2} \leq \frac{h_K^{\frac{5}{2}}}{2\sqrt{30}} ||u''||_\infty \tag{62}$$

Pour calculer l'estimation d'erreur en semi norme $H^1(K)$, remarquons que $f = u - \mathcal{I}_1 u$ est nulle en a, b, et donc en intégrant par partie on a $\int_a^b f'^2 = -\int_a^b f f''$, d'où l'inégalité $\int_a^b f'^2 \leq ||f''||_{L^\infty} \int_a^b |f|$.

ce qui nous donne :

$$|u-\mathcal{I}_1 u|_{H^1} = \|(u-\mathcal{I}_1 u)'\|_{L^2} \leq \frac{h_K^{\frac{3}{2}}}{2\sqrt{3}}|u''|_{L^\infty} \tag{63}$$

Théorème 1.23. *Pour une famille d'espace d'éléments finis X_h définie en (34) au paragraphe 1.6, nous avons les estimations d'erreur suivante :*

$$\forall u \in W^{2,\infty}(\Omega), \qquad |u-\mathcal{I}_h u|_{H^1(\Omega)} \leq \frac{\sqrt{|\Omega|}}{2\sqrt{3}}\, h\, |u''|_{L^\infty(\Omega)}. \tag{64}$$

$$||u-\mathcal{I}_h u||_{L^2(\Omega)} \leq \frac{\sqrt{|\Omega|}}{2\sqrt{30}}\, h^2\, |u''|_{L^\infty(\Omega)}. \tag{65}$$

Démonstration. Il suffit utiliser l'estimation de l'inéquation (63)

$$|u-\mathcal{I}_h u|^2_{H^1(\Omega)} = \sum_{K\in\mathcal{T}_{d,h}} \int_K |(u-\mathcal{I}_h u)'|^2 dx \leq \sum_{K\in\mathcal{T}_{d,h}} \Big(\frac{h_K^{\frac{3}{2}}}{2\sqrt{3}}|u''|_{L^\infty(K)}\Big)^2$$

Donc

$$|u-\mathcal{I}_h u|^2_{H^1(\Omega)} \leq (\sum_{K\in\mathcal{T}_{d,h}} h_K)\Big(\frac{h}{2\sqrt{3}}|u''|_{L^\infty(\Omega)}\Big)^2 \leq (\sqrt{|\Omega|}\frac{h}{2\sqrt{3}}|u''|_{L^\infty(\Omega)})^2$$

On démontre, la seconde inégalité en utilisant la même technique. □

2 Problème Non Linéaire

Le but de ce chapitre est de donnée les algorithmes de bases pour résoudre des problèmes non linéaire en grand dimension.

Ce problème pourra s'écrire généralement sous la forme : trouver u dans U tel que $F(u) = 0$ où F est une fonctionnelle de $U \mapsto V$ et où U et V sont des espaces de Hilbert (souvent de dimension fini).

Dans de nombreux cas le problème correspond à un problème de minimisation c'est à dire : trouver u dans U tel que $u = \arg\min J$ où $J : U \mapsto \mathbb{R}$ est la fonctionnelle à minimiser avec en plus des containtes ou non.

Ces algorithmes sont les méthodes de point fixe, les méthode de type gradient pour les problèmes de minimisation et pour finir la méthode de Newton. La plupart de ces méthodes utilisent le gradient ou la différentielle, donc nous commenceront par introduire ces notions.

2.1 Rappel de Calcul différentielle

La différentielle au sens de Frechet est la généralisation de la notion de dérivé d'une fonction réelle. La différentielle de $F : U \mapsto V$ en u sera la partie linéaire qui approche $h \mapsto F(u+h) - F(u)$ qui sera noté $DF(u) \in \mathcal{L}(U, V)$ cet à dire que l'on a

$$\lim_{h \longrightarrow 0} \frac{||F(u+h) - F(u) - DF(u)h||}{||h||} = 0$$

où U , V sont deux espaces de Banach, et où $\mathcal{L}(U, V)$ est l'ensemble de application linéaire continue de U dans V, et si $A \in \mathcal{L}(U, V)$, $h \in U$, on note Au est image de u par A, donc dans la formule $DF(u)h$ est image de h par $DF(u)$.

Remarque 4. *La différentielle seconde $D^2F(u)$ est donc une application linéaire de $\mathcal{L}(U, V)$ dans V et l'on peut trivialement voir comme une application bilinéaire de $U \times U$ dans V en utilisant l'écriture suivante $(D(DF(u))h_1)h_2 = D^2F(u)(h_1, h2)$, et par induction la différentielle $n^{ième}$ est une application n linéaire de U^n dans V que l'on note $D^n(u)$ et qui correspond à l'application $(h_1, \cdots, h_n) \mapsto D^n(u)(h_1, \cdots, h_n)$*

Voila Quelques astuce pour calculer une différentielle.

Théorème 2.1 (composition)**.** *Soit deux fonctions $F : V \mapsto W$ et $G : U \mapsto V$ alors on $D(F \circ G)(u) = DF(G(u)) \circ DG(u)$ que l'on note généralement $DF(G(u))\, DG(u)$*

Remarque 5. *Si U est un espace de fonctions réelle et si $F(u) = f(u)$ où f est une fonction réelle $\mathcal{C}^1$ alors $DF(u)h = f'(u)h$, mais attention il n'est pas toujours évidant de définir l'espace V associé, c'est souvent pour cela que l'on parle de calcul formel.*

Théorème 2.2. *Soit une fonction linéaire continue $F : U \mapsto V$, la différentielle est $D(F)(u) = F$ et $\forall h \in U, DF(u)h = F(h)$*

Théorème 2.3. *Soit une fonction n linéaire continue* $F : \prod_i U_i \mapsto V$ *notons* $F_i(u_i) = F(u_0, \cdots, u_i, \cdots, u_n)$ *où les* u_j *pour* $j \neq i$ *sont des constantes définissant* F_i *alors* F_i *est linéaire et l'on peut appliquer le théorème précèdent , et* $DF_i(u_i) = F_i$.

Si $G(u) = F(u, \cdots, u)$ *alors* $DG(u)h = \sum_i F(u, \cdots, h, \cdots, u)$ *, où* h *remplace le* $i^{\text{ème}}$ *parameter de* F.

Les opérateurs linéaires classique intervenant dans les formules sont, $\int$, ∂,...

En appliquant, les règles précédente à Si A est un application linéaire symétrique dans une espace de Hilbert munis du produit scalaire $(.,.)$ pour

$$J(u) = \frac{1}{2}(Au, u) - (b, u)$$

alors on a :

$$DJ(u)v = (Au, v) - b(v), \quad \nabla J(u) = Au - b$$

De même pour une fonctionnelle plus compliqué :

$$J(u) = \int_\Omega \sqrt{1 + \nabla u.\nabla u}\, dx$$

on a donc formellement

$$DJ(u)h = \int_\Omega \frac{\nabla u.\nabla h}{\sqrt{1 + \nabla u.\nabla}}\, dx$$

et la différentielle seconde est

$$D^2J(u)(h_1, h_2) = \int_\Omega \frac{(\nabla h_2.\nabla h_1)}{\sqrt{1 + \nabla u.\nabla u}} - \frac{(\nabla h_2.\nabla u)(\nabla h_1.\nabla u)}{\sqrt{1 + \nabla u.\nabla u}^3} dx$$

2.2 Des algorithmes généraux

2.2.1 Methode de point fixe

Idée est de découpé la fonction $F(u) = Au - B(u)$, où A est un opérateur linéaire

Donc l'algorithme de fixe peut s'écrire : Soit u_0 et $n = 0$

$$A\, u^{n+1} = B(u^n), \qquad n = n + 1$$

2.2.2 Methode de Newton

L'algorithme de Newton peut s'écrire : Soit u_0 et $n = 0$

$$DF(u^n)w^n = F(u^n), \qquad u^{n+1} = u^n - w^n, \qquad n = n + 1$$

Maintenant bien souvent, il peut être intéressant de calculer directement u^{n+1} pour des problèmes de conditions aux limites par exemples=

Soit u_0 et $n = 0$

$$DF(u^n)u^{n+1} = DF(u^n)u^n - F(u^n) \qquad n = n + 1$$

2.3 Des algorithmes de minimisation

Le problème est de trouver le minimum de J qui est une fonction de H espace de Hilbert dans $\mathbb{R}$ (hypothèse espace de Hilbert est fondamemental) car pour définir le gradient il faut avoir un moyen de représenter la forme linéaire $DJ(u)$ comme vecteur de H.

Définition 2.1 (Gradient). *Le gradient $\nabla J(u)$ d'une fonction J de H espace de Hilbert dans $\mathbb{R}$ est défini par*

$$(\nabla J(u), v) = DJ(u)v, \quad \forall v \in H \tag{66}$$

en utilisant le Théorème de représentation de Riez.

Remarque 6. *Si H est une espace de dimension fini, alors généralement il est représenté par une base $\mathcal{B} = \{\boldsymbol{e}^i, i = 1, N\}$ et des vecteurs de $\mathbb{R}^N$ et dans ce cas il peut y avoir plusieurs produit scalaire, et donc plusieurs gradients,*

1. *le produit scalaire canonique associer à la base $\mathcal{B}$ est défini par $(u, v) = \sum_{i=1}^N u_i v_i$ où $u = \sum_{i=1}^N u_i \boldsymbol{e}^i$ et $v = \sum_{i=1}^N v_i \boldsymbol{e}^i$, et donc le gradient $\nabla J(u)$ est le vecteur defini par*

$$\nabla J(u) = \sum_{i=1}^{N} (\partial_i J(u))\, \boldsymbol{e}^i, \quad \text{où} \quad \partial_i J(u) = DJ(u)\, \boldsymbol{e}^i \tag{67}$$

   car le matrice de terme $(\boldsymbol{e}^i, \boldsymbol{e}^j)$ est la matrice identité.

2. *Tous les autres produits scalaires sont défini via une matrice symétrique définie positive noté A de taille N. Le produit scalaire s'écrire $(u, v)_A = (Au, v) = (u, Av)$ et donc dans la base $\mathcal{B}$ le gradient $\nabla_A J(u)$ est clairement solution du problème :*

$$A\, \nabla_A J(u) = \nabla J(u), \quad \text{donc} \quad \nabla_A J(u) = A^{-1} \nabla J(u) \tag{68}$$

Remarque 7 (fonctionnelle quadratique)**.** *Soit A un opérateur linéaire elliptique, (symétrique et définie positif) alors*

$$J(v) = \frac{1}{2}(Av, v) - (f, b) \tag{69}$$

est la fonctionnelle à minimiser correspondant la résolution du problème $Au = f$.

Remarque 8 (Méthode des Moindres carres)**.** *Pour résoudre un problème linéaire $Bu = c$ au sens des moindres carrés, revient à minimiser $\frac{1}{2}||Bv - c||^2$ pour $v \in \mathbb{R}^n$ ce qui est équivalent à minimiser*

$$J : v \in \mathbb{R}^n \mapsto J(v) = \frac{1}{2}(Bv, Bv) - (c, Bv) = \frac{1}{2}({}^t BBv, v) - (c, Bv) \tag{70}$$

Définition 2.2 (fonctionnelle coercive). *Une fonctionnelle $J : V \mapsto \mathbb{R}$ défini sur un espace de vectoriel norme V est dite coercive si $\lim_{||v||_V \to \infty} J(v) = +\infty$*

Définition 2.3 (fonctionnelle elliptique). *Une fonctionnelle $J : H \mapsto \mathbb{R}$ défini sur un espace de Hilbert H est dite elliptique si elle continument différentielle sur H et $\exists \alpha \in \mathbb{R}_*^+$telle que*

$$\forall u, v \in H, \qquad (DJ(u) - DJ(v))(u - v) = (\nabla J(u) - \nabla J(v), u - v) \geq \alpha ||u - v|| \tag{71}$$

Remarque 9 (fonctionnelle quadratique)**.** *Une fonctionnelle elliptique est strictement convexe, coercive et vérifie.*

$$\forall u, v \in H, \qquad J(u) - J(v) \geq DJ(v)(u - v) + \frac{\alpha}{2}||u - v||^2 \tag{72}$$

2.3.1 Methode de Gradient

Définition 2.4 (Méthode de Gradient à pas simple). *Soit $n = 0$ et u^0 donné ainsi de une suite de ρ_k*

$$u^{n+1} = u^n - \rho_n \nabla J(u^n), \qquad n = n + 1 \tag{73}$$

Définition 2.5 (Méthode de Gradient à pas optimal). *Soit $n = 0$ et u^0 donné*

$$h^n = \nabla J(u^n), \quad \rho_n = \arg\min_{\rho} J(u^n - \rho h^n); \qquad u^{n+1} = u^n - \rho_n h^n, \qquad n = n + 1 \tag{74}$$

Remarque 10. *Si la fonctionnelle est quadratique et coercive alors le ρ^n optimal dans la direction h^k peut être calculer avec la formule suivante :*

$$\rho_k = -\frac{(\nabla J(h^k) - \nabla J(0), h^k)}{(\nabla J(h^k), h^k)} \tag{75}$$

Le problème de recherche du ρ optimal correspondant à $\arg\min_\rho J(u^n - \rho h^n)$ n'est pas simple, il y a beaucoup heuristique en anglais cette recherche s'appel ≪*line search*≫,

2.4 Des logiciels

A faire

3 Méthodes d'éléments finis P_1 Lagrange

3.1 Formules de Green

Grâce à la densité de $\mathcal{D}(\overline{\Omega})$ dans $H^1(\Omega)$, on peut démontrer la formule de Green pour les fonctions de $H^1(\Omega)$:

$$\forall u \in H^1(\Omega)\ ,\ \forall v \in H^1(\Omega)\ ,\quad \int_\Omega u\,\frac{\partial v}{\partial x_i}\,d\boldsymbol{x} = -\int_\Omega v\,\frac{\partial u}{\partial x_i}\,d\boldsymbol{x} + \int_{\partial\Omega} u\,v\,n_i\,d\sigma\,, \tag{76}$$

où n_i est la $i$ème composante de $\boldsymbol{n}$ la normale extérieur unitaire, et σ est la mesure du bord.

En utilisant les notations du points 1.1 page 4, et toujours grâce à la densité, et en sommant sur les composantes, pour les fonctions vectorielles de $H(\mathrm{div},\Omega) = \{\boldsymbol{v} \in L^2(\Omega)^d / \nabla.\boldsymbol{v} \in L^2(\Omega)\}$, on peut écrire la formule de Stokes avec le gradient ∇ et la divergence $\nabla.$:

$$\forall u \in H^1(\Omega)\ ,\ \forall \boldsymbol{v} \in H(\mathrm{div},\Omega)\ ,\quad \int_\Omega u\,\nabla.\boldsymbol{v}\,d\boldsymbol{x} = -\int_\Omega (\nabla u).\boldsymbol{v}\,d\boldsymbol{x} + \int_{\partial\Omega} u(\boldsymbol{v}.\boldsymbol{n})\,d\sigma\,, \tag{77}$$

La formule précédente avec v et ∇u, nous donne donc :

$$\forall u \in H^2(\Omega)\ ,\ \forall v \in H^1(\Omega)\ ,\quad \int_\Omega (\Delta u)\,v\,d\boldsymbol{x} = -\int_\Omega \nabla u.\nabla v\,d\boldsymbol{x} + \int_{\partial\Omega} v\frac{\partial u}{\partial \boldsymbol{n}}\,d\sigma\,. \tag{78}$$

3.2 Rappel : Forme linéaire, bilinéaire, vecteur , matrice

Soit V un espace vectoriel de dimension finie réel munie d'une base $\{w^i, i = 1, ..., n\}$.

Soit a une forme bilinéaire de V (application de $V \times V \mapsto \mathbb{R}$) et $\ell \in V'$ une forme linéaire de V (application de $V \mapsto \mathbb{R}$).

Alors la résolution du problème : trouver $u \in V$ tel

$$\forall v \in V, \qquad a(u,v) = l(v), \tag{79}$$

est équivalent à la résolution du système linéaire matricielle $n \times n$ suivant : trouver $U = (u_i)_{i=1,n} \in \mathbb{R}^n$ telle que

$$\boldsymbol{A}U = B, \tag{80}$$

Où la matrice A est définie par

$$A = (a_{ij}) \quad \text{avec} \quad \forall (i,j) \in \{1,...,n\}^2 \quad a_{ij} = a(w^j, w^j) \tag{81}$$

et où le second membre B est donné par

$$B = (b_i) \in \mathbb{R}^d, \qquad \text{avec} \quad \forall i \in \{1,...,n\} \quad b_i = \ell(w^i) \tag{82}$$

Pour finir la solution u est égale à

$$u = \sum_{i=1}^{n} u_i w^i. \tag{83}$$

3.3 Espace affine, convexifié, et simplexe

Définition 3.1. *Dans un espace affine réel $\mathcal{A}$, le barycentre de* $(\lambda_i, P_i)_{i=1,n} \in (\mathbb{R} \times A)^n$ *telle que* $\sum_{i=1}^n \lambda_i \neq 0$ *est l'unique point* $G = \mathrm{bar}((\lambda_i, P_i)_{i=1,n}) \in \mathcal{A}$ *telle que*

$$\forall O \in \mathcal{A}, \qquad \sum_{i=1}^{n} \lambda_i \overrightarrow{OP}_i = (\sum_{i=0}^{n} \lambda_i)\overrightarrow{OG}.$$

Et si cette espace affine A est plongé dans un espace vectoriel, alors

$$G = \frac{\sum_{i=0}^n \lambda_i P_i}{\sum_{i=0}^n \lambda_i}$$

Définition 3.2. *Par définition, si f est une fonction affine de $\mathcal{A} \mapsto \mathcal{B}$, alors cette fonction commute avec les barycentres, c'est-à-dire* $f(\mathrm{bar}((\lambda_i, P_i)_{i=1,n})) = \mathrm{bar}((\lambda_i, f(P_i))_{i=1,n})$*, ou si l'espace affine est plongé dans un espace vectoriel, on a*

$$\text{si} \qquad \sum_{i=1}^{n} \lambda_i = 1, \qquad \text{alors} \quad f(\sum_{i=1}^{n} \lambda_i x_i) = \sum_{i=1}^{n} \lambda_i f(x_i) \tag{84}$$

Définition 3.3. *Le convexifié d'un ensemble S de points de $\mathbb{R}^d$ est noté $\mathcal{C}(S)$ est le plus petit convexe contenant S et si l'ensemble est fini (*i.e. $S = \{x^i, i = 1,..,n\}$*) alors nous avons :*

$$\mathcal{C}(S) = \left\{ \sum_{i=1}^{n} \lambda_i x^i : \quad \forall (\lambda_i)_{i=1,..,n} \in \mathbb{R}_+^n, \text{ tel que } \sum_{i=1}^{n} \lambda_i = 1 \right\}$$

.

Définition 3.4. *Un k-simplexe $(P^0, ..., P^k)$ est le convexifié des $k+1$ points de $\mathbb{R}^d$ affine independant (donc $k \leq d$)*

- un sommet est un 0-simplexe,
- une arête ou un segment est un 1-simplexe,
- un triangle est un 2-simplexe,
- un tétraèdre est un 3-simplexe ;

La mesure signée d'un d-simplexe $K = (P^0, ..., P^d)$ en dimension d est donnée par

$$mes(P^0, ..., P^d) = \frac{1}{d!} det \left| \overrightarrow{P^0P^1} \ldots \overrightarrow{P^0P^d} \right| = \frac{-1^d}{d!} det \begin{vmatrix} P_1^0 & \ldots & P_1^d \\ \vdots & \ldots & \vdots \\ P_d^0 & \ldots & P_d^d \\ 1 & \ldots & 1 \end{vmatrix}$$

et le d-simplexe sera dit positif si sa mesure est positive.

les sous p-simplexe d'un d-simplexe sont formés avec $p+1$ points distinctes parmi $(P^0, ..., P^d)$. Les ensembles de k-simplexe

- des sommets seront l'ensemble des $d+1$ points (0-simplexe),
- des arêtes seront l'ensemble des $\frac{(d+1)\times d}{2}$ 1-simplexe ,

— des triangles seront l'ensemble des $\frac{(d+1)\times d\times(d-1)}{6}$ 2-simplexe ,

— des hyperfaces seront l'ensemble des $(d+1)$ hyperfaces ($(d-1)$-simplexe) ,

Définition 3.5. *Les coordonnées barycentriques d'un d-simplex $K=(P^0,...,P^d)$ sont des $d+1$ fonctions affines de $\mathbb{R}^d$ dans $\mathbb{R}$ noté $\lambda_i^K, j=0,...,d$ et sont défini par*

$$\forall x\in\mathbb{R}^d; \qquad x=\sum_{i=0}^{d}\lambda_i^K(x)P^i; \qquad et \qquad \sum_{i=0}^{d}\lambda_i^K(x)=1 \tag{85}$$

on a $\lambda_i^K(P^j)=\delta_{ij}$ où δ_{ij} est le symbole de Kroneker. Et les formules de Cramers nous donnent :

$$\lambda_i^K(x)=\frac{mes(P^0,...,P^{i-1},x,P^{i+1},...,P^d)}{mes(P^0,...,P^{i-1},P^i,P^{i+1},...,P^d)}$$

Le gradient de coordonnée barycentrique est donnée par la formule suivante

$$\nabla\lambda_i^K(x)=\frac{-1^{d+i}}{|K|d!}\overrightarrow{P^{n_{i,1}}P^{n_{i,2}}}\wedge...\ \wedge\overrightarrow{P^{n_{i,1}}P^{n_{i,d}}}$$

où $|K|$ est la mesure signée de K, les $(n_{i,j})_{j=1}^d$ pour i fixé est la suite croissante des nombres sans l'élément i, c'est-à-dire $(n_{i,1},...,n_{i,d})=(0,...,i-1,i+1,...,d)$, où le produit vectoriel est l'application $(d-1)$-linéaire alternée $(R^d)^{d-1}\mapsto R^d$ qui est la généralisation du produit vectoriel dans $\mathbb{R}^d$, et qui est définie par la formule suivante :

$$\forall y\in\mathbb{R}^d; \quad x^1\wedge...\wedge x^{d-1}.y=det|x^1,...,x^{d-1},y|$$

Démonstration.

$$\begin{aligned}
D\lambda_i^K(x)y &= \frac{-1^d}{|K|d!}det\begin{vmatrix} P_1^0 & \dots & P_1^{i-1} & y_1 & P_1^{i+1} & \dots & P_1^d \\ \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\ P_d^0 & \dots & P_d^{i-1} & y_d & P_d^{i+1} & \dots & P_d^d \\ 1 & \dots & 1 & 0 & 1 & \dots & 1 \end{vmatrix} \\
&= \frac{-1^{d-d+i}}{|K|d!}det\begin{vmatrix} P_1^0 & \dots & P_1^{i-1} & P_1^{i+1} & \dots & P_1^d & y_1 \\ \vdots & \dots & \vdots & \vdots & \dots & \vdots & \vdots \\ P_d^0 & \dots & P_d^{i-1} & P_d^{i+1} & \dots & P_d^d & y_d \\ 1 & \dots & 1 & 1 & \dots & 1 & 0 \end{vmatrix} \\
&= \frac{-1^{d-d+i}}{|K|d!}det\begin{vmatrix} P_1^{n_{i,1}} & \dots & P_1^{n_{i,d}} & y_1 \\ \vdots & \dots & \vdots & \vdots \\ P_d^{n_{i,1}} & \dots & P_d^{n_{i,d}} & y_d \\ 1 & \dots & 1 & 0 \end{vmatrix} \\
&= \frac{-1^{i}}{|K|d!}det\begin{vmatrix} P_1^{n_{i,1}} & \overrightarrow{P_1^{n_{i,1}}P_1^{n_{i,2}}} & \dots & \overrightarrow{P_1^{n_{i,1}}P_1^{n_{i,d}}} & y_1 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ P_d^{n_{i,1}} & \overrightarrow{P_d^{n_{i,1}}P_d^{n_{i,2}}} & \dots & \overrightarrow{P_d^{n_{i,1}}P_d^{n_{i,d}}} & y_d \\ 1 & 0 & \dots & 0 & 0 \end{vmatrix} \\
&= \frac{-1^{i+d}}{|K|d!}\overrightarrow{P^{n_{i,1}}P^{n_{i,2}}}\wedge...\ \wedge\overrightarrow{P^{n_{i,1}}P^{n_{i,d}}}.y
\end{aligned}$$

□

Donc si $d = 2$ nous avons donc

$$\nabla \lambda_i^K(x) = \frac{-1^i}{2|K|} \overrightarrow{P^{n_{i,1}} P^{n_{i,2}}}^{\perp}$$

où $\overrightarrow{PQ}^{\perp}$ est la rotation de $\pi/2$ du vecteur $\overrightarrow{PQ}$. Ce qui donne

$$\nabla \lambda_0^K(x) \frac{+1}{2|K|} = \overrightarrow{P^1 P^2}^{\perp}, \qquad \nabla \lambda_1^K(x) \frac{-1}{2|K|} = \overrightarrow{P^0 P^2}^{\perp}, \qquad \nabla \lambda_2^K(x) \frac{+1}{2|K|} = \overrightarrow{P^0 P^1}^{\perp}$$

ce que l'on peut réécrire comme avec la convention $P^3 = P^0$ et $P^4 = P^1$:

$$\nabla \lambda_i^K(x) = \frac{+1}{2|K|} (\overrightarrow{P^{i+1} P^{i+2}})^{\perp}, \tag{86}$$

Et si $d = 3$ alors on a

$$\nabla \lambda_i^K(x) = \frac{-1^{i+1}}{6|K|} \overrightarrow{P^{n_{i,1}} P^{n_{i,2}}} \wedge \overrightarrow{P^{n_{i,1}} P^{n_{i,3}}}$$

avec la numérotation des sommets des faces suivante : $(n_{0,j})_{j=1,2,3} = (1, 2, 3)$, $(n_{1,j})_{j=1,2,3} = (0, 2, 3)$, $(n_{2,j})_{j=1,2,3} = (0, 1, 3)$, et $(n_{3,j})_{j=1,2,3} = (0, 1, 2)$.

Pour supprimer le -1^{i+1} il suffit d'utiliser la numérotation des sommets des faces suivante :

$$(\tilde{n}_{0,j})_{j=1,2,3} = (2, 1, 3), \qquad (\tilde{n}_{1,j})_{j=2,1,3} = (0, 2, 3), \tag{87}$$

$$(\tilde{n}_{2,j})_{j=1,2,3} = (1, 0, 3), \qquad (\tilde{n}_{3,j})_{j=1,2,3} = (0, 1, 2). \tag{88}$$

Ce qui donne :

$$\nabla \lambda_i^K(x) = \frac{1}{6|K|} \overrightarrow{P^{\tilde{n}_{i,1}} P^{\tilde{n}_{i,2}}} \wedge \overrightarrow{P^{\tilde{n}_{i,1}} P^{\tilde{n}_{i,3}}} \tag{89}$$

Remarque 11. *Sur le d-simplex référence note $\hat{K} = (0, \boldsymbol{e}_1, \cdots, \boldsymbol{e}_d)$ où les $\boldsymbol{e}_1, \cdots, \boldsymbol{e}_d)$ forment la base canonique de $\mathbb{R}^d$ alors les coordonnes barycentrique de $\lambda_i^{\hat{K}}$ sont*

$$\lambda_0^{\hat{K}}(\hat{\boldsymbol{x}}) = 1 - \sum_{i=1}^{d} \hat{x}_i, \qquad et \quad \lambda_j^{\hat{K}}(\hat{\boldsymbol{x}}) = \hat{x}_j \quad pour\ j = 1, \cdots, d \tag{90}$$

où $\boldsymbol{x} = (x_i)_{i=0,\cdot,d}$.

Les coordonnées barycentriques permettent de construire une fonction affine à partir des valeurs aux sommets de simplexe K via le lemme suivant :

Lemme 3.6. *Soit f une fonction affine d'un espace affine $\mathcal{A}$ de dimension d à valeur dans espace vectoriel réel et un d-simplexe $K = (P^0, ..., P^d)$ de $\mathcal{A}$ alors on à l'égalité suivante :*

$$f(x) = \sum_{i=0}^{d} f(P^i)\, \lambda_i^K(x). \tag{91}$$

Démonstration. Il suffit juste de remarquer que comme f est affine, elle commute avec les barycentres et que x est le barycentres de $(\lambda_i^K(x), P^i), i = 0, \cdots, d$ car les $(P^0, ..., P^d$ forme une base affine de $\mathcal{A}$ □

3.4 Formule d'intégration

Let D be a N-dimensional bounded domain. For an arbitrary polynomials f of degree r, if we can find particular points $\vec{\xi}_j$, $j = 1, \cdots, J$ in D and constants ω_j such that

$$\int_D f(\vec{x}) = |D| \sum_{\ell=1}^{L} \omega_\ell f(\vec{\xi}_\ell) \tag{92}$$

then we have the error estimation (see Crouzeix-Mignot (1984)), and then there exists a constant $C > 0$ such that,

$$\left| \int_D f(\vec{x}) - |D| \sum_{\ell=1}^{L} \omega_\ell f(\vec{\xi}_\ell) \right| \leq C|D|h^{r+1} \tag{93}$$

3.4.1 Formule d'intégration sur le triangle

Soit le triangle $K = [q^{i_0} q^{i_1} q^{i_2}]$ ($d = 2$, le point intégration est défini via le point $\hat{\xi}^\ell$ sur le triangle référence $\hat{K} = \{(0,0), (1,0), (0,1)\}$, comme suit

$$\xi_l = (1 - \sum_{k=0}^{d} \hat{\xi}_k) q^{i_0} + \sum_{k=1}^{d} \hat{\xi}_k) q^{i_k}$$

L	points $\hat{\xi}^\ell$ in $\hat{K}$	ω_ℓ	degree of exact
1	$\left(\frac{1}{3}, \frac{1}{3}\right)$	$\lvert T_k \rvert$	1
3	$\left(\frac{1}{2}, \frac{1}{2}\right)$	$\lvert T_k \rvert/3$	2
	$\left(\frac{1}{2}, 0\right)$	$\lvert T_k \rvert/3$	
	$\left(0, \frac{1}{2}\right)$	$\lvert T_k \rvert/3$	
7	$\left(\frac{1}{3}, \frac{1}{3}\right)$	$0.225\lvert T_k \rvert$	5
	$\left(\frac{6-\sqrt{15}}{21}, \frac{6-\sqrt{15}}{21}\right)$	$\frac{(155-\sqrt{15})\lvert T_k \rvert}{1200}$	
	$\left(\frac{6-\sqrt{15}}{21}, \frac{9+2\sqrt{15}}{21}\right)$	$\frac{(155-\sqrt{15})\lvert T_k \rvert}{1200}$	
	$\left(\frac{9+2\sqrt{15}}{21}, \frac{6-\sqrt{15}}{21}\right)$	$\frac{(155-\sqrt{15})\lvert T_k \rvert}{1200}$	
	$\left(\frac{6+\sqrt{15}}{21}, \frac{6+\sqrt{15}}{21}\right)$	$\frac{(155+\sqrt{15})\lvert T_k \rvert}{1200}$	
	$\left(\frac{6+\sqrt{15}}{21}, \frac{9-2\sqrt{15}}{21}\right)$	$\frac{(155+\sqrt{15})\lvert T_k \rvert}{1200}$	
	$\left(\frac{9-2\sqrt{15}}{21}, \frac{6+\sqrt{15}}{21}\right)$	$\frac{(155+\sqrt{15})\lvert T_k \rvert}{1200}$	
3	$(0,0)$	$\lvert T_k \rvert/3$	1
	$(1,0)$	$\lvert T_k \rvert/3$	
	$(0,1)$	$\lvert T_k \rvert/3$	

Voilà une formule bien utile d'ou son nom :

Proposition 3.7. *(formule magique) Pour calculer, l'integral sur un d-simplex K du produit de puissance entiere $n_i >= 0$ des coordonnées barycentrique λ_i de ce simplex, il suffit utiliser la formule suivante :*

$$\int_K \prod_{i=0}^{d} \lambda_i^{n_i} = |K| \frac{d! \prod_{i=0}^{d} n_i!}{(d + \sum_{i=0}^{d} n_i)!} \tag{94}$$

Et donc

$$\int_K \lambda_i = |K| \frac{1}{(d+1)}, \quad et \quad \int_K \lambda_i \lambda_j = |K| \frac{1 + \delta_{ij}}{(d+1)(d+2)} \tag{95}$$

Démonstration. Le démonstration ce fait par récurrence sur la dimension de l'espace d sur le simplex de référence $\hat{K}_d = (0, \boldsymbol{e}_1, \cdots, \boldsymbol{e}_d)$ où les $\boldsymbol{e}_i$ sont les vecteurs de base canonique de $\mathbb{R}^d$.

— Pour $d = 1$, il faut montrer que

$$\int_0^1 x^{n_1}(1-x)^{n_0} = \frac{n_0! n_1!}{(1 + n_0 + n_1)!} \tag{96}$$

si $n_1 = 0$ c'est vrai, puis il suffit de faire une récurrence sur n_0,pour tout n_1. L'héritage, quelque soit $n_1 >= 0$, on suppose la propriété vraie pour n_0 quelque soit $n_1 >= 0$ en intégrant par partie et en utilisant l'héritage pour $n_1 + 1$ et n_0 on a :

$$\int_0^1 x^{n_1}(1-x)^{n_0+1} = \frac{n_0+1}{n_1+1} \int_0^1 x^{n_1+1}(1-x)^{n_0} = \frac{n_0+1}{n_1+1} \frac{(n_1+1)! n_0!}{(2 + n_0 + n_1)!} = \frac{n_1!(n_0+1)!}{(2 + n_0 + n_1)!}$$

ce qui fini le cas $d = 1$.

— Supposons la formule vraie pour $d-1$, et montrons la formule pour d. Nous avons ajouter une dimension en nous avons en utilisants Fubini

$$\int_{\hat{K}_d} \prod_{i=0}^{d} \lambda_i^{n_i} = \int_0^1 \lambda_d^{n_d} \left(\int_{(1-x_d)\hat{K}} \prod_{i=0}^{d-1} \lambda_i^{n_i} dx_1 \cdots dx_{d-1} \right) dx_d \tag{97}$$

Où, $(1 - x_d)K_{d-1}$ est intersection de K_d avec l'hyperlan de dernière composante égala à x_d, et il suffit de remarquer que la mesure de $(1 - x_d)\hat{K}_{d-1}$ est égale à $(1 - x_d)^{d-1}|\hat{K}_{d-1}|$ et que les $\lambda_i^{\hat{K}_d}{}_{|(1-x_d)K_{d-1}} = (1 - x_d)\lambda_i^{(1-x_d)K_{d-1}}$ pour $i \leq d$.
Ceci nous donne donc :

$$\int_{\hat{K}_d} \prod_{i=0}^{d} \lambda_i^{n_i} = \int_0^1 \lambda_d^{n_d}(1 - x^d)^{d-1+\sum_{i=0}^{d-1} n_i} \left(\int_{\hat{K}_{d-1}} \prod_{i=0}^{d-1} \lambda_i^{n_i} dx_1 \cdots dx_{d-1} \right) dx_d \tag{98}$$

En appliquant l'hypothèse de récurrence et en remarquons que $\lambda_d^{\hat{K}_d} = x_d$ on a :

$$\int_{\hat{K}_d} \prod_{i=0}^{d} \lambda_i^{n_i} = |\hat{K}_{d-1}| \frac{(d-1)! \prod_{i=0}^{d-1} n_i!}{(d - 1 + \sum_{i=0}^{d-1} n_i)!} \int_0^1 x_d^{n_d}(1 - x^d)^{d-1+\sum_{i=0}^{d-1} n_i} dx_d \tag{99}$$

Il suffit appliquer la formule dans le cas $d = 1$, et $|K_d| = |K_{d-1}|/d$ pour avoir

$$\int_{\hat{K}_d} \prod_{i=0}^{d} \lambda_i^{n_i} = |\hat{K}_d| \frac{d(d-1)! \prod_{i=0}^{d-1} n_i!}{(d-1+\sum_{i=0}^{d-1} n_i)!} \frac{n_d!(d-1+\sum_{i=0}^{d-1} n_i)!}{(d+\sum_{i=0}^{d} n_i)!} \tag{100}$$

En regroupe les termes et simplifiant les termes on obtient les résultats.

□

3.5 Le maillage

Commençons par définir la notion de maillage simplexial.

Définition 3.8. *Un maillage simplexial $\mathcal{T}_{d,h}$ d'un ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^d$ est un ensemble de $d-$ simplex K^k de $\mathbb{R}^d$ pour $k = 1, N_t$ (triangle si $d = 2$ et tétraèdre si $d = 3$), tel que l'intersection de deux d-simplex distincts $\overline{K}^i, \overline{K}^j$ de $\mathcal{T}_{d,h}$ soit :*

- *l'ensemble vide,*
- *ou p-simplex commun à K et K' avec $p \leq d$*

Le maillage $\mathcal{T}_{d,h}$ couvre l'ouvert défini par :

$$\mathcal{O}_h \overset{def}{=} \overset{\circ}{\overline{\bigcup_{K \in \mathcal{T}_{d,h}} K}} \tag{101}$$

De plus, $\mathcal{T}_{0,h}$ désignera l'ensemble des sommets de $\mathcal{T}_{d,h}$ et $\mathcal{T}_{1,h}$ l'ensemble des arêtes de $\mathcal{T}_{d,h}$ et l'ensemble de faces serat $\mathcal{T}_{d-1,h}$. Le bord $\partial\mathcal{T}_{d,h}$ du maillage $\mathcal{T}_{d,h}$ est défini comme l'ensemble des faces qui ont la propriété d'appartenir à un unique d-simplex de $\mathcal{T}_{d,h}$. Par conséquent, $\partial\mathcal{T}_{d,h}$ est un maillage du bord $\partial\mathcal{O}_h$ de $\mathcal{O}_h$. Par abus de langage, nous confondrons une arête d'extrémités (a, b) et le segment ouvert $]a, b[$, ou fermé $[a, b]$.

3.6 Condition aux Limites de type Dirichlet

La question relativement simple, peut écrire simplement d'une point de vue algorithmique la prise en couple de Condition au limite de Dirichlet.

D'un point de vue formel , nous avons a résoudre numériquement un problème affine. qui s'écrit comme suit :

Trouver $u \in V_g = g + V_0$ telle que

$$\forall v \in V_0; \quad \boldsymbol{a}(u, v) = l(v) \tag{102}$$

Où g est la représentation de condition aux limites de Dirichlet, V_0 est l'espace discret des fonctions avec limites de Dirichlet homogène (nulle), $\boldsymbol{a}$ la forme bilinéaire du problème et l la forme linéaire.

On notera V l'espace discret associé sans condition aux limites de Dirichlet, et soit $\mathcal{B} = \{w^i, \in \mathcal{I}\}$ la base associer de V , de plus la base $\mathcal{B}_0 = \{w^i, \in \mathcal{I}_0\}$ de V_0 est telle que $\mathcal{I}_0 \subset \mathcal{I}$.

Généralement, on a $g = \sum_{j\in\mathcal{I}\setminus\mathcal{I}_0} g_j w^i$ et $g_i = 0$ pour $i \in \mathcal{I}_0$, notre problème est trouver le vecteur u_j tel que $u = \sum_{j\in\mathcal{I}} u_j w^j$ and

$$\left.\begin{array}{llr} \forall i \in \mathcal{I}_0 & \displaystyle\sum_{j\in\mathcal{I}} a(w^j, w^i)u_j & = l(w^i) \\ \forall i \in \mathcal{I}\setminus\mathcal{I}_0 & u_i & = g_i \end{array}\right\} \quad (103)$$

Nous pouvons encore écrire notre problème comme suit :

$$\left.\begin{array}{ll} (AU = B)_i, & i \in \mathcal{I}_0 \\ (U = G)_i, & i \notin \mathcal{I}_0 \end{array}\right\} \quad (104)$$

où $A = (a_{ij})$ avec $a_{ij} = \boldsymbol{a}(w^j, w^i)$ et où $B = (b_i)$ avec $b_i = l(w^i)$, et $G = (g_i)$.

La méthode mathématique est résoudre le problème le suivant trouver les u_i pour $i \in \mathcal{I}_0$, telle que

$$\sum_{j\in\mathcal{I}_0} a(w^j, w^i)u_j = l(w^i) - \sum_{j\in\mathcal{I}\setminus\mathcal{I}_0} a(w^j, w^i)g_j \quad (105)$$

que l'on peut réécrire comme suit

$$(\tilde{A}U = B - AG)_i, \quad i \in \mathcal{I}_0 \quad (106)$$

où $\tilde{A}$ est la matrice A carre restreint à $\mathcal{I}_0$.

Notons I_0 la matrice diagonal de coefficient i égal à 1 si $i \in \mathcal{I}_0$ et à 0 si $i \in \mathcal{I}\setminus\mathcal{I}_0$ et notons $I_\gamma = Id - I_0$ le complement à l'identité et remarquons que $I_0 I_\gamma = 0$.

Nous pouvons réécrire, le problème comme : Trouver U solution de

$$(I_0 A I_0 + I\gamma)U = I_0(B - AI_0G) + I_\gamma G \quad (107)$$

La seule difficulté est cette méthode de résolution est qu'elle dur à programmer. Donc voici des méthodes facile à programmer.

3.6.1 Méthode de pénalisation exacte

La méthode de pénalisation par ε est : trouvez le vecteur $U_\varepsilon \in \mathbb{R}^{\mathcal{I}}$ solution du problème

$$(A + \frac{1}{\varepsilon}I_\gamma)U_\varepsilon = I_0 B + \frac{1}{\varepsilon}I_\gamma G, \quad i \in \mathcal{I}_0 \quad (108)$$

On peut montrer facilement que pour ε suffisamment petit qu'il existe une constante $C > 0$ telle que

$$||U - U_\varepsilon|| \leq \varepsilon C \quad (109)$$

A partir de (108) et (107) en mutilipiant à droit par I_0, on a :

$$\begin{array}{rcl} I_0 A I_0 U_\varepsilon & = & I_0(B - AI_\gamma U_\varepsilon) \\ I_0 A I_0 U & = & I_0(B - AI_\gamma G) \end{array}$$

comme $I_0 A I_0$ est inversible sur l'image de I_0 qui est V_0 , on a donc

$$||I_0(U_\varepsilon - U)|| \leq ||(I_0 A I_0 + I_\gamma)^{-1}|| \, ||A|| \, ||I_\gamma(U_\varepsilon - G)|| \tag{110}$$

d'où

$$||U_\varepsilon - U|| \leq ||I_0(U_\varepsilon - U)|| + ||I_\gamma(U_\varepsilon - U)|| \leq C_0 ||I_\gamma(U_\varepsilon - G)|| \tag{111}$$

Mais maintenant, à partir de (108) en mutilipiant à gauche par I_γ et comme $I_\gamma I_0 = 0$, on a :

$$I_\gamma(U_\varepsilon - G) = \varepsilon(-I_\gamma A U_\varepsilon + I_\gamma I_0 B) = -\varepsilon I_\gamma A U_\varepsilon \tag{112}$$

donc en utilisant $||U_\varepsilon|| \leq ||U|| + ||U_\varepsilon - U||$, (111) et (112) on a

$$||U_\varepsilon - U|| \leq \varepsilon(C_0 ||A||(||U|| + ||U_\varepsilon - U||) \tag{113}$$

Pour ε est suffisamment petit on a

$$||U_\varepsilon - U|| \leq \varepsilon \frac{C_0 ||A||}{1 - \varepsilon C_0 ||A||} ||U||. \tag{114}$$

ce qui prouve l'inégalité (109).

Numériquement les nombres ont seulement 16 chiffres significatifs en double précision, c'est-à-dire que si l'on prend $\frac{1}{\varepsilon} = \mathtt{tgv} = 10^{30}$ nous travaillons dans des espaces de nombre indépendant est l'erreur est de l'ordre de 10^{-30}. Cela fonctionne numériquement car la pénalisation n'apparait que sur la diagonal.

Niveau implémentation, il suffit d'ajoute les lignes suivantes :

Pour $\mathtt{i} \in \mathcal{I} \setminus \mathcal{I}_0$ faire

```
a_ii = tgv ;
b_i  = tgv * g_i
```

Attention dans les méthodes itérative de types gradiant conjugue, la première étape de l'algorithme défini les conditions ou limites, si le vecteur initiale ne contient pas déjà les valeurs de c'est condition aux limites, il donc faire plus de 1 itérations, voir modification de Gradient conjugue comme suit :

```
if (fabs(g2) < reps2)
 if ( !iter  && fabs(g2)>1e-30 && epsold >0 ) {
              //    change eps: converge to fast due to the penalization of BC.
   reps2 =  epsold*epsold*fabs(g2);
   cout << "CG converge to fast (pb of BC)  restart: " << iter << "  ro = " << ro
        << " ||g||^2 = " << g2 << " <= " << reps2 << " new eps2 ="<< reps2 <<endl;
     }
else  {
   cout << iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
   return 1;                                                  //    ok, convergence
       }
```

3.6.2 Condition de Dirichlet dans les méthodes de minimisation (GC, GMRES, ...)

Si la méthode est base sur la minimisation de $||Ax - b||_C$, alors généralement et si la méthode n'utilise que le produit matrice vecteur , alors

il suffit d'ecrire comme produit matrice vecteur le produit $I_0 Ax$ et de imposer comme second membre $I_0 B$. la condition aux limite sera donné dans le vecteur d'initialisation de l'algorithme, ce qui est très simple a programmer.

3.7 Le problème et l'algorithme

Le problème est de résoudre numériquement l'équation de la chaleur dans un domaine Ω de $\mathbb{R}^2$.

$$-\Delta u = f, \quad \text{dans } \Omega, \quad u = g \quad \text{sur } \partial\Omega. \tag{115}$$

Nous utiliserons la discrétisation par des éléments finis P_1 Lagrange construite sur un maillage $\mathcal{T}_{d,h}$ de Ω. Notons V_h l'espace des fonctions éléments finis et V_{0h} les fonctions de V_h nulle sur bord de Ω_h (ouvert obtenu comme l'intérieur de l'union des triangles fermés de $\mathcal{T}_{d,h}$).

$$V_h = \{v \in \mathcal{C}^0(\Omega_h) / \forall K \in \mathcal{T}_{d,h}, v|_K \in P_1(K)\} \tag{116}$$

Après utilisation de la formule de Green, et en multipliant par v, le problème peut alors s'écrire :

Calculer $u_h^{n+1} \in V_h$ à partir de u_h^n , où la donnée initiale u_h^0 est interpolé P_1 de u_0.

$$\int_\Omega \nabla u_h \nabla v_h = \int_\Omega f v_h, \quad \forall v_h \in V_{0h}, \tag{117}$$

$$u_h = g \quad \text{sur les sommets de } \mathcal{T}_{d,h} \text{ dans } \partial\Omega$$

Nous nous proposons de programmer cette méthode l'algorithme du gradient conjugue pour résoudre le problème linéaire car nous avons pas besoin de connaître explicitement la matrice, mais seulement, le produit matrice vecteur.

Puis, notons, $(w^i)_{i=1,N_s}$ les fonctions de base de V_h associées aux N_s sommets de $\mathcal{T}_{d,h}$ de coordonnées $(q^i)_{i=1,N_s}$, tel que $w^i(q_j) = \delta_{ij}$. Notons, U_i le vecteur de $\mathbb{R}^{N_s}$ associé à u et tel que $u = \sum_{i=1,N_s} U_i w^i$.

Sur un triangle K formé de sommets i, j, k tournants dans le sens trigonométrique. Notons, H_K^i le vecteur hauteur pointant sur le sommet i du triangle K et de longueur l'inverse de la hauteur, alors on a comme dans (86) :

$$\nabla w^i|_K = H_K^i = \frac{(\overrightarrow{q^j q^k})^\perp}{2\, aire_K} \tag{118}$$

où l'opérateur $\perp$ de $\mathbb{R}^2$ est défini comme la rotation de $\pi/2$, ie. $(a, b)^\perp = (-b, a)$.

Algorithme 1.

Le programme d'éléments finis sans matrice

1. *Soit la matrice* $\boldsymbol{M}_{\alpha,\beta}$ *associée la forme bilinéaire*

$$\boldsymbol{a}_{\alpha,\beta}(u,v) = \int_\Omega \alpha uv + \int_\Omega \beta \nabla u.\nabla v$$

et la matrice élémentaire 3×3 $\boldsymbol{M}^K$ *associée la forme bilinéaire*

$$\boldsymbol{a}^K_{\alpha,\beta}(u,v) = \int_K \alpha uv + \int_K \beta \nabla u.\nabla v$$

sur l'espace des fonctions $P_1(K)$ *dans la base* $\lambda_i^K, i = 0,1,2$.

2. *Pour la méthde du gradient conjugue nous avons juste besoin de la méthode* `addMatMul` *qui ajoute à* y *le produit matrice vecteur* Ax, *c'est à dire :*

$$y+ = \boldsymbol{M}_{\alpha,\beta} x$$

Ce cas s'écrit mathématiquement :
Pour $K \in \mathcal{T}_{d,h}$, *pour* $i = 0,1,2$ *et pour* $j = 0,1,2$ *faire :*

$$y_{i_i^K} + = \begin{cases} \boldsymbol{M}^K_{ij}\ x_{i_j^K} & si\ i_i^K \notin \Gamma \\ 0 & sinon \end{cases}$$

3. *On peut calculer* $\boldsymbol{b} = (\int_\Omega w^i f_h)_i = \boldsymbol{M}_{1,0} \boldsymbol{f}_h$ *en utilisant la formule où* $\boldsymbol{f}_h$ *est le vecteur de la fonction* f_h *qui est l'interpolé de* f , *c'est a dire* $f_h = \sum_i f(x_i) w^i$ *et* $\boldsymbol{f}_h = (f(q_i))_i$.
4. *La matrice* A *est simplement* $M_{0,1}$.

Pour finir, on initialiserons le gradient conjugué avec l'initilisation suivante :

$$\boldsymbol{x} = (x_i)_i = \begin{cases} 0 & si\ i\ n'est\ pas\ sur\ le\ bord \\ g(q_i) & si\ i\ est\ sur\ le\ bord \end{cases}$$

pour resoudre de système linéaire

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

avec la matrice $\boldsymbol{A} = \boldsymbol{M}_{0,1}$

3.8 La résolution d'un systéme linéaire avec le gradient conjugué

L'algorithme du gradient conjugué présenté dans cette section est utilisé pour résoudre le système linéaire $Ax = b$, où A est une matrice symétrique positive $n \times n$. Cet algorithme est basé sur la minimisation de la fonctionnelle quadratique $E : \mathbb{R}^n \to \mathbb{R}$ suivante :

$$E(x) = \frac{1}{2}(Ax, x) - (b, x),$$

La version préconditionnée avec un matrice C correspond après avec un changement de variable $x = Cy$ à la resolution de ${}^tCACy = {}^tCb$, et à la minimisation de la fonctionnelle :

$$E_c(y) = \frac{1}{2}(ACy, y)_C - (b, y)_C,$$

où $(.,.)_C$ est le produit scalaire associer à une une matrice C, symétrique définie positive de $\mathbb{R}^n$. et car la matrice CAC est symétrique positive.

Algorithme 2.

Le gradient conjugué preconditionné

soient $x^0 \in \mathbb{R}^n$, ε, C donnés
$G^0 = Ax^0 - b$
$H^0 = -CG^0$
si $(G^0, G^0)_C < \varepsilon$ *stop*
— *pour* $i = 0$ *à* n
$\rho = -\frac{(G^i, H^i)}{(H^i, AH^i)}$
$x^{i+1} = x^i + \rho H^i$
$G^{i+1} = G^i + \rho AH^i$
$\gamma = \frac{(G^{i+1}, G^{i+1})_C}{(G^i, G^i)_C}$
$H^{i+1} = -CG^{i+1} + \gamma H^i$
si $(G^{i+1}, G^{i+1})_C < \varepsilon$ *stop*

Théorème 3.9. *Notons* $\mathcal{E}_C(x) = \sqrt{(Ax - \overline{x}, x - \overline{x})_C}$, *l'erreur dans la norme de A preconditionné, où* $\overline{x}$ *est la solution du problème. Alors l'erreur à l'intération k un gradient conjugué est majoré :*

$$\mathcal{E}_C(x^k) \leq 2 \left(\frac{\sqrt{K_C(A) - 1}}{\sqrt{K_C(A) + 1}} \right)^k \mathcal{E}_C(x^0)$$

où $K_C(A)$ *est le conditionnement de la matrice* CA, *c'est à dire* $K_C(A) = \frac{\lambda_1^C}{\lambda_n^C}$ *où* λ_1^C *(resp.* λ_n^C*) est la plus petite (resp. grande) valeur propre de matrice* CA .

Le démonstration est technique et est faite dans [Lascaux et Théodor, chap 8.3]. Voila comment écrire un gradient conjugué avec ces classes.

3.8.1 Gradient conjugué préconditionné

Listing 1: *(GC.hpp)*

```
               //    exemple de programmation du gradient conjugue precondiconnée
template<class R,class M,class P>
int GradienConjugue(const M & A,const P & C,const KN_<R> &b,KN_<R> &x,
                    int nbitermax,double eps)
{
   int n=b.N();
   assert(n==x.N());
   KN<R> g(n), h(n), Ah(n), & Cg(Ah);          //    on utilise Ah pour stocke Cg
   g = A*x;
   g -= b;                                                          //    g = Ax-b
   Cg = C*g;                                        //    gradient preconditionne
   h =-Cg;
   R g2 = (Cg,g);
   R reps2 = eps*eps*g2;                                     //    epsilon relatif
   for (int iter=0;iter<=nbitermax;iter++)
     {
       Ah = A*h;
       R ro =  - (g,h)/ (h,Ah);          //    ro optimal (produit scalaire usuel)
       x += ro *h;
       g += ro *Ah;      //    plus besoin de Ah, on utilise avec Cg optimisation
       Cg = C*g;
       R g2p=g2;
       g2 = (Cg,g);
       cout << iter << "  ro = " << ro << " ||g||^2 = " << g2 << endl;
       if (g2 < reps2) {
          cout << iter << "  ro = " << ro << " ||g||^2 = " << g2 << endl;
          return 1;                                                    //    ok
          }
       R gamma = g2/g2p;
       h *= gamma;
       h -= Cg;                                              //    h = -Cg * gamma* h
     }
   cout << " Non convergence de la méthode du gradient conjugue " <<endl;
   return 0;
}
//    la matrice Identite --------------------
template <class R>
class MatriceIdentite: VirtualMatrice<R> { public:
 typedef  VirtualMatrice<R>::plusAx plusAx;
 MatriceIdentite(int n):VirtualMatrice<R> (n) {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {   Ax+=x; }
 plusAx operator*(const KN<R> &  x) const {return plusAx(this,x);}
};
```

3.8.2 Test du gradient conjugué

Pour finir voilà, un petit programme pour tester le GC sur cas différent. Le troisième cas étant la résolution de l'équation différencielle : $-u'' = 1$ sur $[0,1]$ avec comme conditions aux limites $u(0) = u(1) = 0$, par la méthode de l'élément fini. La solution exact est $f(x) = x(1-x)/2$, nous verifions donc l'erreur sur le resultat.

Listing 2: *(GradConjugue.cpp)*

```
#include <fstream>
#include <cassert>
#include <algorithm>

using namespace std;

#define KN_CHECK
#include "RNM.hpp"
#include "GC.hpp"

typedef double R;
class MatriceLaplacien1D: VirtualMatrice<R> { public:
 MatriceLaplacien1D(int n) :VirtualMatrice<R>(n) {};
 void addMatMul(const KN_<R> & x, KN_<R> & Ax) const ;
 plusAx operator*(const KN<R> &  x) const {return plusAx(*this,x);}
};

 void MatriceLaplacien1D::addMatMul(const KN_<R> & x, KN_<R> & Ax) const {
  int n= x.N(),n_1=n-1;
  double h=1./(n_1), h2= h*h, d = 2/h, d1 = -1/h;
  R Ax0=Ax[0], Axn_1=Ax[n_1];
  Ax=0;
  for (int i=1;i< n_1 ; i++)
    Ax[i] = (x[i-1] +x[i+1]) * d1 + x[i]*d  ;
                                                                    //    CL
   Ax[0]=x[0];
   Ax[n_1]=x[n_1];
  }


int main(int argc,char ** argv)
{
 typedef KN<double> Rn;
 typedef KN_<double> Rn_;
 typedef KNM<double> Rnm;
 typedef KNM_<double> Rnm_;
 {
  int n=10;
  Rnm A(n,n),C(n,n),Id(n,n);
  A=-1;
  C=0;
  Id=0;
  Rn_ Aii(A,SubArray(n,0,n+1));        //   la diagonal de la matrice A sans copy
  Rn_ Cii(C,SubArray(n,0,n+1));        //   la diagonal de la matrice C sans copy
  Rn_ Idii(Id,SubArray(n,0,n+1));     //   la diagonal de la matrice Id sans copy
```

```
  for (int i=0;i<n;i++)
     Cii[i]= 1/(Aii[i]=n+i*i*i);
  Idii=1;
  cout << A;
  Rn x(n),b(n),s(n);
  for (int i=0;i<n;i++) b[i]=i;
  cout << "GradienConjugue preconditionne par la diagonale " <<  endl;
  x=0;
  GradienConjugue(A,C,b,x,n,1e-10);
  s = A*x ;
  cout << " solution : A*x= "  << s << endl;
  cout << "GradienConjugue preconditionnee par la identity  " << endl;
  x=0;
  GradienConjugue(A,MatriceIdentite<R>(n),b,x,n,1e-6);
  s = A*x ;
  cout << s << endl;
 }
 {
  cout << "GradienConjugue laplacien 1D  par la identity  " << endl;
  int N=100;
  Rn b(N),x(N);
  R h= 1./(N-1);
  b= h;
  b[0]=0;
  b[N-1]=0;
  x=0;
  R t0=CPUtime();
  GradienConjugue(MatriceLaplacien1D(n),MatriceIdentite<R>(n) ,b,x,N,1e-5);
  cout << " Temps cpu = " << CPUtime() - t0<< "s" << endl;
  R err=0;
  for (int i=0;i<N;i++)
    {
    R xx=i*h;
    err= max(fabs(x[i]- (xx*(1-xx)/2)),err);
    }
  cout << "Fin err=" << err << endl;
 }
 return 0;
}
```

3.8.3 Sortie du test

```
10x10   :
          10  -1  -1  -1  -1  -1  -1  -1  -1  -1
          -1  11  -1  -1  -1  -1  -1  -1  -1  -1
          -1  -1  18  -1  -1  -1  -1  -1  -1  -1
          -1  -1  -1  37  -1  -1  -1  -1  -1  -1
          -1  -1  -1  -1  74  -1  -1  -1  -1  -1
          -1  -1  -1  -1  -1 135  -1  -1  -1  -1
          -1  -1  -1  -1  -1  -1 226  -1  -1  -1
          -1  -1  -1  -1  -1  -1  -1 353  -1  -1
          -1  -1  -1  -1  -1  -1  -1  -1 522  -1
          -1  -1  -1  -1  -1  -1  -1  -1  -1 739
```

```
        GradienConjugue preconditionne par la diagonale
6  ro = 0.990712 ||g||^2 = 1.4253e-24
 solution : A*x= 10     :    1.60635e-15  1 2 3 4 5 6 7 8 9

GradienConjugue preconditionnee par la identity
9  ro = 0.0889083 ||g||^2 = 2.28121e-15
10      :       6.50655e-11    1 2 3 4 5 6 7 8 9

GradienConjugue laplacien 1D  preconditionnee par la identity
48  ro = 0.00505051 ||g||^2 = 1.55006e-32
 Temps cpu = 0.02s
Fin err=5.55112e-17
```

Exercice 8.

Modifier, l'exemple `GradConjugue.cpp`, pour résoudre le problème suivant, trouver $u(x,t)$ une solution

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = f; \quad \text{dans }]0,L[$$

$$\text{pour } t=0, u(x,0)=u_0(x) \quad \text{et } u(0,t)=u(L,t)=0$$

en utilisant un θ schéma pour discrétiser en temps, c'est à dire que

$$\frac{u^{n+1}-u^n}{\Delta t} - \frac{\partial^2\big(\theta u^{n+1} + (1-\theta)u^n\big)u}{\partial x^2} = f; \quad \text{dans }]0,L[$$

où u^n est une approximation de $u(x, n\Delta t)$, faite avec des éléments finis P_1, avec un maillage régulier de $]0,L|$ en M éléments. les fonctions élémentaires seront noté w^i, avec pour $i=0,...,M$, avec $x_i = \frac{i/N}{L}$

$$w^i|_{]x_{i-1},x_i[\cup]0,L[} = \frac{x-x_i}{x_{i-1}-x_i}, \quad w^i|_{]x_i,x_{i+1}[\cup]0,L[} = \frac{x-x_i}{x_{i+1}-x_i}, \quad w^i|_{]0,L[\setminus]x_{i-1},x_{i+1}[} = 0$$

C'est a dire qu'il faut commencer par construire du classe qui modélise la matrice

$$\mathcal{M}_{\alpha\beta} = \left(\int_{]O,L[} \alpha w^i w^j + \beta (w^i)'(w^j)'\right)_{i=1...M, j=1...M}$$

en copiant la classe `MatriceLaplacien1D`.
Puis, il suffit, d'approcher le vecteur $F = (f_i)_{i=0..M} = \left(\int_{]O,L[} f w^i\right)_{i=0..M}$ par le produit $F_h = \mathcal{M}_{1,0}\,\big(f(x_i)\big)_{i=1,M}$, ce qui revient à remplacer f par

$$f_h = \sum_i f(x_i) w^i$$

.

Q1 Ecrire l'algorithme mathematiquement, avec des matrices

Q2 Trancrire l'algorithme

Q3 Visualiser les résultat avec gnuplot, pour cela il suffit de crée un fichier par pas de temps contenant la solution, stocker dans un fichier, en inspirant de

```
#include<sstream>
#include<ofstream>
#include<iostream>
  ....
  stringstream ff;
  ff << "sol-"<< temps << ends;
  cout << " ouverture du fichier" <<ff.str.c_str()<< endl;
  {
     ofstream file(ff.str.c_str());
     for (int i=0;i<=M;++i)
       file <<x[i]<< endl;
   }          //    fin de bloc => destruction de la variable file
                                             //    => fermeture du fichier
  ...
```

4 Algorithmique

4.1 Introduction

Dans ce chapitre, nous allons décrire les notions d'algorithmique élémentaire, La notions de complexité.

Puis nous présenterons les chaînes et de chaînages ou liste d'abord d'un point de vue mathématique, puis nous montrerons par des exemples comment utiliser cette technique pour écrire des programmes très efficaces et simple.

Rappelons qu'une chaîne est un objet informatique composée d'une suite de maillons. Un maillon, quand il n'est pas le dernier de la chaîne, contient l'information permettant de trouver le maillon suivant. Comme application fondamentale de la notion de chaîne, nous commencerons par donner une méthode efficace de construction de l'image réciproque d'une fonction.

Ensuite, nous utiliserons cette technique pour construire l'ensemble des arêtes d'un maillage, pour trouver l'ensemble des triangles contenant un sommet donné, et enfin pour construire la structure creuse d'une matrice d'éléments finis.

4.2 Complexité algorithmique

Copié de http://fr.wikipedia.org/wiki/Complexité algorithmique

La théorie de la complexité algorithmique s'intéresse à l'estimation de l'efficacité des algorithmes. Elle s'attache à la question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?

Dans les années 1960 et au début des années 1970, alors qu'on en était à découvrir des algorithmes fondamentaux (tris tels que quicksort, arbres couvrants tels que les algorithmes de Kruskal ou de Prim), on ne mesurait pas leur efficacité. On se contentait de dire : « cet algorithme (de tri) se déroule en 6 secondes avec un tableau de 50 000 entiers choisis au hasard en entrée, sur un ordinateur IBM 360/91. Le langage de programmation PL/I a été utilisé avec les optimisations standard ». (cet exemple est imaginaire)

Une telle démarche rendait difficile la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé, etc.

Une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série The Art of Computer Programming. Il complétait cette analyse de considérations propres à la théorie de l'information : celle-ci par exemple, combinée à la formule de Stirling, montre qu'il ne sera pas possible d'effectuer un tri général (c'est-à-dire uniquement par comparaisons) de N élements en un temps croissant moins rapidement avec N que N ln N sur une machine algorithmique (à la différence peut-être d'un ordinateur quantique).

Pour qu'une analyse ne dépende pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur, il faut utiliser comme unité de comparaison des « opérations élémentaires » en fonction de la taille des données en entrée. Exemples d'opérations élémentaires : accès à une cellule mémoire, comparaison de valeurs, opérations arithmétiques

(sur valeurs à codage de taille fixe), opérations sur des pointeurs. Il faut souvent préciser quelles sont les opérations élémentaires pertinentes pour le problème étudié : si les nombre manipulés restent de taille raisonnable, on considérera que l'addition de deux entiers prend un temps constant, quels que soient les entiers considérés (ils seront en effet codés sur 32 bits). En revanche, lorsque l'on étudie des problèmes de calcul formel où la taille des nombres manipulés n'est pas bornée, le temps de calcul du produit de deux nombre dépendra de la taille de ces deux nombres.

On définit alors la taille de la donnée sur laquelle s'applique chaque problème par un entier lié au nombre d'éléments de la donnée. Par exemple, le nombre d'éléments dans un algorithme de tri, le nombre de sommets et d'arcs dans un graphe.

On évalue le nombre d'opérations élémentaires en fonction de la taille de la donnée : si 'n' est la taille, on calcule une fonction t(n).

Les critères d'analyse : le nombre d'opérations élémentaires peut varier substantiellement pour deux données de même taille. On retiendra deux critères :

- analyse au sens du plus mauvais cas : t(n) est le temps d'exécution du plus mauvais cas et le maximum sur toutes les données de taille n. Par exemple, le tri par insertion simple avec des entiers présents en ordre décroissants.
- analyse au sens de la moyenne : comme le « plus mauvais cas » peut en pratique n'apparaître que très rarement, on étudie tm(n), l'espérance sur l'ensemble des temps d'exécution, où chaque entrée a une certaine probabilité de se présenter. L'analyse mathématique de la complexité moyenne est souvent délicate. De plus, la signification de la distribution des probabilités par rapport à l'exécution réelle (sur un problème réel) est à considérer.

On étudie systématiquemnt la complexité asymptotique, noté grâce aux notations de Landau.

idée 1 : évaluer l'algorithme sur des données de grande taille. Par exemple, lorsque n est 'grand', $3n^3 + 2n^2$ est essentiellement $3n^3$.

idée 2 : on élimine les constantes multiplicatrices, car deux ordinateurs de puissances différentes diffèrent en temps d'exécution par une constante multiplicatrice. De $3 * n^3$, on ne retient que n^3

L'algorithme est dit en $0(n^3)$.

L'idée de base est donc qu'un algorithme en $0(n^a)$ est « meilleur » qu'un algorithme en $0(n^b)$ si $a < b$.

Les limites de cette théorie :

le coefficient multiplicateur est oublié : est-ce qu'en pratique $100 * n^2$ est « meilleur » que $5*n^3$? l'occupation mémoire, les problèmes d'entrées/sorties sont occultés, dans les algorithmes numériques, la précision et la stabilité sont primordiaux. Point fort : c'est une notion indispensable pour le développement d'algorithmes efficaces.

Les principales classes de complexité :

- logarithmique : $\log_b n$
- linéaire : $an + b$
- polynomiale : $\sum_{i=0}^{n} a_i n^i$
- exponentielle : a^n
- factorielle : $n!$

Pour finir, il est aussi possible de parlé de la complexité mémoire, d'un algorithme.

4.3 Base, tableau, couleur

Recherche de la valeur maximale d'un tableau de `double` `u` de taille `N`.

```
double umax=u[0];
for (int i=1;i<N;++i)
  umax=max(umax,u[i]);
```

Si l'on veut connaitre l'indice `imax` associé à la plus grand valeur

```
int imax=0;
for (int i=1;i<N;++i)
  if(u[imax] < u[i])
    imax=i;
```

Exercice 9. Ecrire un programme trouve les 5 plus grande valeurs du tableau `u` avec une complexité de $O(N)$ en temps calcul et $O(1)$ en mémoire additionnelle.

4.3.1 Décalage d'un tableau

Nous voulons décaler le tableau de 1 comme suit formellement $u^{new}[i-1] = u^{old}[i], \forall i \in \{1, .., n-1\}$.

```
for (int i=1;i<N;++i)
  u[i-1]=u[i];
```

Remarque, tout ce passe bien car la valeur de $u[i-1]$ n'est plus utilisé dans la boucle, pour les i suivants mais dans le cas contraire $u^{new}[i+1] = u^{old}[i], \forall i \in \{0, .., n-2\}$, il suffit de faire la boucle à l'envers pour éviter le problème d'écrasement.

```
for (int i=N-1; i>1 ;--i )
  u[i]=u[i-1];
```

Ou si l'on ne veut pas changer le sens de parcours, alors il suffit de stocker les dépendances dans une deux mémoires auxiliaires en $u[i-1]$ et $u[i]$, ce qui donne

```
u0=u[0];                          //    valeur précédente avant modification
for (int i=1;i<N;++i)
  {
  u1=u[i];                                  //    valeur avant modification
  u[i]=u0;
  u0=u1;                          //    valeur précédente avant modification
  }
```

4.3.2 Renumérote un tableau

Soit sigma une permutation (bijection) de $\{0, ..., n-1\}$, le but est de changer l'ordre du tableau par la permutation σ ou par la permutation inverse σ^-1.

Il est difficile de faire ce type algorithme sur place, mais avec une copie, c'est trivail :

```
                                                         //    remumérotation
for (int i=0; i<n;++i )
  v[i]=u[sigma[i]];
```

```
                                                   //    remumérotation inverse
for (int i=0; i<n;++i )
  v[sigma[i]]=u[i];                           //    remarquons v[sigma[i]]=u[i];
```

Et donc pour construction la permutation inverse stocker dans le tableau sigma1 il suffit d'ecrire :

```
                                                     //    permutation inverse
for (int i=0; i<n;++i )
  sigma1[sigma[i]]=i;
```

4.4 Construction de l'image réciproque d'une fonction

On va montrer comment construire l'image réciproque d'une fonction F. Pour simplifier l'exposé, nous supposerons que F est une fonction entière de $I = \{0, ..., n-1\}$ dans $J = \{0, ..., m-1\}$ et que ses valeurs sont stockées dans un tableau. Le lecteur pourra changer les bornes du domaine de définition ou de l'image sans grand problème.

Voici une méthode simple et efficace pour construire $F^{-1}(j)$ pour de nombreux j dans $\{0, ..., m-1\}$, quand n et m sont des entiers raisonnables. Pour chaque valeur $j \in Im\, F \subset \{0, ..., m-1\}$, nous allons construire la liste de ses antécédents. Pour cela nous utiliserons deux tableaux : `int head_F[m]` contenant les "têtes de listes" et `int next_F[n]` contenant la liste des éléments des $F^{-1}(i)$. Plus précisément, si i_1, i_2,...,$i_p \in [0, n]$, avec $p \geq 1$, sont les antécédents de j, head_F[j]$=i_p$, next_F[i_p]$=i_{p-1}$, next_F[i_{p-1}]$=i_{p-2}$, $\ldots$, next_F[i_2]$=i_1$ et next_F[i_1]= -1 (pour terminer la chaîne).

L'algorithme est découpé en deux parties : l'une décrivant la construction des tableaux next_F et head_F, l'autre décrivant la manière de parcourir la liste des antécédents.

Algorithme 3. **Construction de l'image réciproque d'un tableau**

1. *Construction :*

```
int Not_In_I = -1;
 for (int j=0;j<m;j++)
  head_F[j]= Not_In_I;              //    initialement, les listes
                                   //    des antécédents sont vides
 for (int i=0;i<n;i++)
   {j=F[i];next_F[i]=head_F[j];head_F[j]=i;}  //   chaînage amont
```

2. *Parcours de l'image réciproque de j dans $[0, n]$:*

```
 for (int i=head_F[j];i!= Not_In_I;i=next_F[i])
  { assert(F[i]==j);          //    j doit être dans l'image de i
                                         //    ... votre code
  }
```

Exercice 10. *Le pourquoi est laissé en exercice.*

4.5 Construction de classe d'equivalence

Le problème est très simple, nous avons une relation équivalence sur un ensemble d'entiers $I = \{0, \cdots, n-1\}$, qui n'est défini que par ensemble de pair m d'entier noté a et qui est prolongé par transitivité. Pour compter le nombre de classe d'équivalence, où le nombre de composante connexe du graphe associé, un algorithme naturelle et trivial est en $n \times m$ et codé comme suit

```
int nbClassEquivalenceBrute(int n,int m,int (* a)[2])
 {
    vector<int> c(n);
    int nc = n ;                                //    nombre de composante
    for(int i=0;i<n;++i) ce[i]=i;
    for(int j=0, j<m; ++j)
      {
        int i1= a[j][0], i2=a[j][1];
        int c1=c[i1],c2=c[i2];      //    Attention il faut utiliser une copie
                                   //    car le tableau c va est modifié
        if(c1 !=c2)
         {
          nc--;
          for(int i= 0; i< n; ++i)
              if(c[i] == c2) c[i] = c1;
         }
      }
      return nc;
 }
```

Maintenant pour accélérer l'algorithme, nous allons associer un arbre à chaque composante et donc en chaque noeud on allons associer le père pour descendre dans l'arbre, et la racine qui n'a pas de père nous associons aucune valeur (code avec une valeur négative strict).

Donc pour trouver la racine r de l'arbre passant par i, il suffit d'écrire

```
int r = i;
while ( arbre[r] >=0) r=arbre[r];
```

Le coup calcul est majoré par la profondeur de l'arbre.

Pour fusionner les deux arbres de racine $r1$ et $r1$ et de profondeur respective $p1$ et $p2$, il suffit de écrire

— $arbre[r1] = r2$ et la racine commune sera $r2$ et la profondeur $max(p1 + 1, p2)$, ou bien
— $arbre[r2] = r1$ et la racine commune sera $r1$ et la profondeur $max(p1, p2 + 1)$,

si l'on stocke pour les racines une valeur négative donnant la profondeur de l'arbre associer, on va pouvoir fusionner les arbres pour obtenir l'arbre de profondeur minimal afin de limiter le coût calcul. Il est facile de montrer que la profondeur majore par $log_2(n)$.

Voici un façon simple de coder cette algorithme :

```
int nbClassEquivalence(int n,int m,int (* a)[2])
 {
    vector<int> arbre(n);
    int nc = n ;                    //    nombre de composante = nombre de noeud
    for(int i=0;i<n;++i)
      arbre[i]=-1;  //    tous les arbre sont réduit à un noeud de profondeur 1

    for(int j=0, j<m; ++j)
      {
        int i1= a[j][0], i2=a[j][1], r1=i1,r2=i2,rr1,rr2;
        while ( (rr1=arbre[r1]) >=0) r1=rr1;
        while ( (rr2=arbre[r2]) >=0) r2=rr2;
        if(r1 !=r2)          //    racine différente, on fusionne les 2 arbres
         {
           nc--;                                 //    un composante de moins
           if( rr1<rr2) arbre[r2]=r1;          //    r1 plus profond : r2->r1
           else if( rr2<rr1) arbre[r1]=r2;     //    r2 plus profond : r1->r2
           else {arbre[r2]=r1;                   //    même profondeur r2->r1
                 arbre[r1]--;}          //   et la profondeur de r1 croît de 1
          }
      }
    return nc;
 }
```

De faite on a programmé l'algorithme de Kruskal `https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal` pour construire l'arbre de poids minimal si les pairs a ont été triés par coût croissant.

4.6 Tri par tas (heap sort)

La fonction HeapSort voir http://fr.wikipedia.org/wiki/Tri_par_tas pour les explications du trie par tas. La complexité de ce trie est $O(nlog_2(n))$.

```
template<class T>
void  HeapSort(T *c,long n)
{                                                              //    starting 0...
  long l,j,r,i;
  T crit;
  if( n <= 1) return;
  l = n/2;
  r = n-1;
  while (1) {                                                       //    label 2
    if(l <= 0 ) {                                                  //    label 20
      crit = c[r];
      c[r--] = c[0];
    if ( !r  ) { c[0]=crit; return;}
    } else  crit = c[--l];
    j=l;
    while (1) {                                                     //    label 4
      i=j;
      j=2*j+1;
      if  (j>r) {c[i]=crit;break;}                                 //    L8 -> G2
      if ((j<r) && (c[j] < c[j+1])) j++;                               //    L5
      if (crit < c[j]) c[i]=c[j];                                   //    L6+1 G4
      else {c[i]=crit;break;}                                       //    L8 -> G2
    }
  }
}
```

4.7 Construction des arêtes d'un maillage

Rappelons qu'un maillage est défini par la donnée d'une liste de points et d'une liste d'éléments (des triangles par exemple). Dans notre cas, le maillage triangulaire est implémenté dans la classe Mesh qui a deux membres nv et nt respectivement le nombre de sommets et le nombre de triangle, et qui a l'opérateur fonction (int j,int i) qui retourne le numero de du sommet i du triangle j. Cette classe Mesh pourrait être par exemple :

```
  class Mesh { public:
   int nv,nt;                                  //    nb de sommet, nb de triangle
   int (* nu)[3];                                                //    connectivité
   double (* c)[2];                                    //    coordonnées de sommet
   int operator()(int i,int j) const { return nu[i][j]);}
   Mesh(const char * filename);                        //    lecture d'un maillage
 }
```

Ou bien sur la classe défini en **??**.

Dans certaines applications, il peut être utile de construire la liste des *arêtes du maillage*, c'est-à-dire l'ensemble des arêtes de tous les éléments. La difficulté dans ce type de construction réside

dans la manière d'éviter – ou d'éliminer – les doublons (le plus souvent une arête appartient à deux triangles).

Nous allons proposer deux algorithmes pour déterminer la liste des arêtes. Dans les deux cas, nous utiliserons le fait que les arêtes sont des segments de droite et sont donc définies complètement par la donnée des numéros de leurs deux sommets. On stockera donc les arêtes dans un tableau `arete[nbex][2]` où `nbex` est un majorant du nombre total d'arêtes. On pourra prendre grossièrement `nbex = 3*nt` ou bien utiliser la formule d'Euler en 2D

$$\texttt{nbe} = \texttt{nt} + \texttt{nv} + \texttt{nb_de_trous} - \texttt{nb_composantes_connexes}, \tag{119}$$

où `nbe` est le nombre d'arêtes (*edges* en anglais), nt le nombre de triangles et nv le nombre de sommets (*vertices* en anglais).

La première méthode est la plus simple : on compare les arêtes de chaque élément du maillage avec la liste de *toutes* les arêtes déjà répertoriées. Si l'arête était déjà connue on l'ignore, sinon on l'ajoute à la liste. Le nombre d'opérations est $nbe * (nbe + 1)/2$.

Avant de donner le première algorithme, indiquons qu'on utilisera souvent une petite routine qui échange deux paramètres :

```
template<class T> inline void Exchange (T& a,T& b) {T c=a;a=b;b=c;}
```

Algorithme 4. **Construction lente des arêtes d'un maillage $\mathcal{T}_{d,h}$**

```
int ConstructionArete(const Mesh & Th,int  (* arete)[2])
{
  int SommetDesAretes[3][2] = { {0,1},{1,2},{2,0}};
  nbe = 0;                                        //    nombre d'arête;
  for(int t=0;t<Th.nt;t++)
   for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]);
      int  j= Th(t,SommetDesAretes[et][1]);
      if (j < i) Exchange(i,j)                 //    on oriente l'arête
      bool existe =false;      //    l'arête n'existe pas a priori
       for (int e=0;e<nbe;e++)    //   pour les arêtes existantes
        if (arete[e][0] == i && arete[e][1] == j)
          {existe=true;break;}  //    l'arête est déjà construite
      if (!existe)                                //    nouvelle arête
        arete[nbe][0]=i,arete[nbe++][1]=j;}
   return nbe;
}
```

Cet algorithme trivial est bien trop cher (en $O(9\,n^2)$) dès que le maillage a plus de 10^3 sommets (plus de 9 10^6 opérations). Pour le rendre de l'ordre du nombre d'arêtes, on va remplacer la boucle sur l'ensemble des arêtes construites par une boucle sur l'ensemble des arêtes ayant le même plus petit numéro de sommet. Dans un maillage raisonnable, le nombre d'arêtes incidentes sur un sommet est petit, disons de l'ordre de six, le gain est donc important : nous obtiendrons ainsi un algorithme en $9 \times nt$.

Pour mettre cette idée en oeuvre, nous allons utiliser l'algorithme de parcours de l'image réciproque de la fonction qui à une arête associe le plus petit numéro de ses sommets. Autrement dit, avec les notations de la section précédente, l'image par l'application F d'une arête

sera le minimum des numéros de ses deux sommets. De plus, la construction et l'utilisation des listes, c'est-à-dire les étapes 1 et 2 de l'algorithme 3 seront simultanées.

Algorithme 5.

Construction rapide des arêtes d'un maillage $\mathcal{T}_{d,h}$

```
int ConstructionArete(const Mesh & Th,int  (* arete)[2],
                   int nbex) {
   int SommetDesAretes[3][2] = { {1,2},{2,0},{0,1}};
   int end_list=-1;
   int * head_minv = new int [Th.nv];
   int * next_edge = new int [nbex];

   for ( int i =0;i<Th.nv;i++)
     head_minv[i]=end_list;                          //   liste vide

   nbe = 0;                                          //    nombre d'arête;

   for(int t=0;t<Th.nt;t++)
    for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]);  //    premier sommet
;
      int  j= Th(t,SommetDesAretes[et][1]); //    second sommet ;
      if (j < i) Exchange(i,j)            //    on oriente l'arête
      bool existe =false;      //    l'arête n'existe pas a priori
       for (int e=head_minv[i];e!=end_list;e = next_edge[e] )
                    //    on parcourt les arêtes déjà construites
        if ( arete[e][1] == j)  //    l'arête est déjà construite
          {existe=true;break;}                      //    stop
      if (!existe) {                           //    nouvelle arête
        assert(nbe < nbex);
        arete[nbe][0]=i,arete[nbe][1]=j;
                                //    génération des chaînages
        next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;}
      }
  delete []  head_minv;
  delete []  next_edge;
  return nbe;
}
```

Preuve : la boucle `for(int e=head_minv[i];e!=end_list;e=next_edge[e])` permet de parcourir toutes des arêtes (i, j) orientées $(i < j)$ ayant même i, et la ligne :

```
next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;
```

permet de chaîner en tête de liste des nouvelles arêtes. Le `nbe++` incrémente pour finir le nombre d'arêtes. ∎

Remarque : les sources sont disponible : `http://www.ann.jussieu.fr/~hecht/ftp/cpp/mesh-liste/BuildEdges.cpp`.

Exercice 11. *Il est possible de modifier l'algorithme précédent en supprimant le tableau* `next_edge` *et en stockant les chaînages dans* `arete[i][0]`*, mais à la fin, il faut faire une boucle de plus sur les sommets pour reconstruire* `arete[.][0]`*.*

Exercice 12. *Construire le tableau `adj` d'entier de taille 3 `nt` qui donne pour l'arête `p=i+3k` (arête `i` du triangle `k`) , qui donne l'arête correspondante `p'=i'+3k'` .*

- *si cette arete est interne alors `adj[i+3k]=i'+3k'` où est l'arete `i'` du triangle `k'`, remarquons : `i'=adj[i+3k]%3, k'=adj[i+3k]/3`.*
- *sinon `adj[i+3k]=-1`.*

4.8 Construction des triangles contenant un sommet donné

La structure de données classique d'un maillage permet de connaître directement tous les sommets d'un triangle. En revanche, déterminer tous les triangles contenant un sommet n'est pas immédiat. Nous allons pour cela proposer un algorithme qui exploite à nouveau la notion de liste chaînée.

Rappelons que si `Th` est une instance de la class `Mesh` (voir 4.7), `i=Th(k,j)` est le numéro global du sommet `j`$\in [0,3[$ de l'élément `k`. L'application F qu'on va considérer associe à un couple (k,j) la valeur `i=Th(k,j)`. Ainsi, l'ensemble des numéros des triangles contenant un sommet `i` sera donné par les premières composantes des antécédents de `i`.

On va utiliser à nouveau l'algorithme 3, mais il y a une petite difficulté par rapport à la section précédente : les éléments du domaine de définition de F sont des *couples* et non plus simplement des entiers. Pour résoudre ce problème, remarquons qu'on peut associer de manière unique au couple `(k,j)`, où $j \in [0,m[$, l'entier `p(k,j)=k*m+j` [1]. Pour retrouver le couple `(k,j)` à partir de l'entier `p`, il suffit d'écrire que `k` et `j` sont respectivement le quotient et le reste de la division euclidienne de `p` par `m`, autrement dit :

$$\mathtt{p} \longrightarrow (\mathtt{k},\mathtt{j}) = (\mathtt{k} = \mathtt{p}/\mathtt{m}, \mathtt{j} = \mathtt{p}\%\mathtt{m}). \tag{120}$$

Voici donc l'algorithme pour construire l'ensemble des triangles ayant un sommet en commun :

1. Noter au passage que c'est ainsi que `C++` traite les tableaux à double entrée : un tableau `T[n][m]` est stocké comme un tableau à simple entrée de taille `n*m` dans lequel l'élément `T[k][j]` est repéré par l'indice `p(k,j) = k*m+j`.

Algorithme 6.

Construction de l'ensemble des triangles ayant un sommet commun

Préparation :

```
int end_list=-1,
int *head_s = new int[Th.nv];
int *next_p = new int[Th.nt*3];
int i,j,k,p;
for (i=0;i<Th.nv;i++)
  head_s[i] = end_list;
for (k=0;k<Th.nt;k++)                       //     forall triangles
  for(j=0;j<3;j++) {
    p = 3*k+j;
    i = Th(k,j);
    next_p[p]=head_s[i];
    head_s[i]= p;}
```

Utilisation : parcours de tous les triangles ayant le sommet numéro `i`

```
  for (int p=head_s[i]; p!=end_list; p=next_p[p])
  {
     k=p/3;
     j = p % 3;
     assert( i == Th(k,j));
                                            //     votre code
  }
```

Exercice 13. *Optimiser le code en initialisant* `p =-1` *et en remplaçant* `p = 3*j+k` *par* `p++`*.*

Remarque : les sources sont disponible : `http://www.ann.jussieu.fr/~hecht/ftp/cpp/mesh-liste/BuildListeTrianglesVertex.cpp`.

Remarque : en utilisant la STL, il est facile de construire un programme répondant a la question. La Construction en $O(nt\, log(nt))$ opération

```
vector<vector<int> > lst(Th.nv);
for (int k=0;k<Th.nt;++k)
  for(int j=0;j<3;++j)
    lst[Th(k,j)].push_back(k);
```

Utilisation pour parcours optimal de tous les triangles ayant le sommet numéro `i`

```
for (int l=0;l<lst[i].size();++l)
  {
   int k= lst[i][l];
   assert( Th(k,0) == i || Th(k,1) == i || Th(k,2) == i );
   ...
  }
```

4.9 Construction de la structure d'une matrice morse

Il est bien connu que la méthode des éléments finis conduit à des systèmes linéaires associés à des matrices très *creuses*, c'est-à-dire contenant un grand nombre de termes nuls. Dès que le maillage est donné, on peut construire le graphe des coefficients *a priori* non nuls de la matrice. En ne stockant que ces termes, on pourra réduire au maximum l'occupation en mémoire et optimiser les produits matrices/vecteurs.

4.9.1 Description de la structure morse

La structure de données que nous allons utiliser pour décrire la matrice creuse est souvent appelée "matrice morse" (en particulier dans la bibliothèque MODULEF), dans la littérature anglo-saxonne on trouve parfois l'expression "Compressed Row Sparse matrix" (cf. SIAM book...). Notons `n` le nombre de lignes et de colonnes de la matrice, et `nbcoef` le nombre de coefficients non nuls *a priori*. Trois tableaux sont utilisés : `a[k]` qui contient la valeur du k-ième coefficient non nul avec $k \in [0, nbcoef[$, `ligne[i]` qui contient l'indice dans `a` du premier terme de la ligne `i+1` de la matrice avec $i \in [-1, n[$ et enfin `colonne[k]` qui contient l'indice de la colonne du coefficient `k∈[0:nbcoef[`. On va de plus supposer ici que la matrice est symétrique, on ne stockera donc que sa partie triangulaire inférieure. En résumé, on a :

$$\mathtt{a}[\mathtt{k}] = a_{\mathtt{ij}} \quad \text{pour } \mathtt{k} \in [\mathtt{ligne}[\mathtt{i}-1]+1, \mathtt{ligne}[\mathtt{i}]] \quad \text{et } \mathtt{j} = \mathtt{colonne}[\mathtt{k}] \quad \text{si } \mathtt{i} \leq \mathtt{j}$$

et s'il n'existe pas de k pour un couple `(i,j)` ou si $\mathtt{i} \leq \mathtt{j}$ alors $a_{\mathtt{ij}} = 0$.

La classe décrivant une telle structure est :

```
class MatriceMorseSymetrique {
  int n,nbcoef;  //    dimension de la matrice et nombre de coefficients non nuls
  int *ligne,* colonne;
  double *a;
  MatriceMorseSymetrique(Maillage & Th);                        //    constructeur
  double*  pij(int i,int j) const ;     //    retourne le pointeur sur le coef i,j
                                                  //    de la matrice si il existe
}
```

Exemple : on considère la partie triangulaire inférieure de la matrice d'ordre 10 suivante (les valeurs sont les rangs dans le stockage et non les coefficients de la matrice) :

$$\begin{pmatrix}
0 & . & . & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . \\
. & 2 & 3 & . & . & . & . & . & . & . \\
. & 4 & 5 & 6 & . & . & . & . & . & . \\
. & . & 7 & . & 8 & . & . & . & . & . \\
. & . & . & 9 & 10 & 11 & . & . & . & . \\
. & . & 12 & . & 13 & . & 14 & . & . & . \\
. & . & . & . & . & 15 & 16 & 17 & . & . \\
. & . & . & . & 18 & . & . & . & 19 & . \\
. & . & . & . & . & . & . & . & . & 20
\end{pmatrix}$$

On numérote les lignes et les colonnes de `[0..9]`. On a alors :

```
n=10,nbcoef=20,
ligne[-1:9] = {-1,0,1,3,6,8,11,14,17,19,20};
colonne[21] ={0, 1,  1,2,  1,2,3,  2,4,  3,4,5,  2,4,6,
               5,6,7,  4,8,  9};
a[21]                    //    ... valeurs des 21 coefficients de la matrice
```

4.9.2 Construction de la structure morse par coloriage

Nous allons maintenant construire la structure morse d'une matrice symétrique à partir de la donnée d'un maillage d'éléments finis P_1.

Les coefficient a_{ij} non nulles de la matrice sont tels qu'il existe un triangle K contenant les sommets i et j.

Donc, pour construire la ligne i de la matrice, il faut trouver tous les sommets j tels que i, j appartiennent à un même triangle. Ainsi, pour un noeud donné i, il s'agit de lister les sommets appartenant aux triangles contenant i. Le premier ingrédient de la méthode sera donc d'utiliser l'algorithme 6 pour parcourir l'ensemble des triangles contenant i. Mais il reste une difficulté : il faut éviter les doublons. Nous allons pour cela utiliser une autre technique classique de programmation qui consiste à « colorier »les coefficients déjà répertoriés : pour chaque sommet i (boucle externe), nous effectuons une boucle interne sur les triangles contenant i puis une balayage des sommets j de ces triangles en les coloriant pour éviter de compter plusieurs fois les coefficients a_{ij} correspondant. Si on n'utilise qu'une couleur, nous devons remettre la couleur du initial les sommets avant de passer à un autre i. Pour éviter cela, nous allons utiliser plusieurs couleurs, et on changera simplement de couleur de marquage à chaque fois qu'on changera de sommet i dans la boucle externe car toutes couleurs utilisées ont un numéro strictement inférieur.

Algorithme 7. **Construction de la structure d'une matrice morse**

```
MatriceMorseSymetrique::MatriceMorseSymetrique(const Mesh & Th){
  int i,j,jt,k,p,t;
  n = Th.nv;                                  //    nombre de ligne
  int color=0, * mark;
  mark = new int [n];    //    pour stocker la couleur d'un sommet
                        //    initialisation du tableau de couleur
  for(j=0;j<Th.nv;j++)  mark[j]=color;
  color++;                                 //    on change la couleur
     //   construction optimisee de l'image réciproque: i=Th(k,j)
  int end_list=-1,*head_s,*next_t;
  head_s = new int [Th.nv];
  next_p = new int [Th.nt*3];
  int i,j,k,p=0;
  for (i=0;i<Th.nv;i++)
    head_s[i] = end_list;
  for (k=0;k<Th.nt;k++)
   for(j=0;j<3;j++)
     { next_p[p]=head_s[i=Th(k,j)]; head_s[i]=p++;}
//   1) calcul du nombre de coefficients non nuls
//   a priori de la matrice pour pourvoir faire les allocations
  nbcoef = 0;
  for(i=0; i<n; i++,color++,nbcoef++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
       for(jt=0; jt< 3 ; jt++ )
        if(i <= (j=Th(t,jt)) && mark[j] != color)) //   nouveau j
        { mark[j]=color; nbcoef++;}      //    => marquage + ajout
//    2) allocations mémoires
  ligne = new int [n+1];
  ligne++;                     //    car le tableau commence en -1;
  colonne = new int [ nbcoef];
  a = new double [nbcoef];
//    3) constructions des deux tableaux ligne et colonne
  ligne[-1] = -1;
  nbcoef =0;
  for(i=0; i<n; ligne[i++]=nbcoef, color++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
       for(jt=0; jt< 3 ; jt++ )
        if ( i <= (j=Th(t,jt)))  && mark[j] != color))
                    //    nouveau coefficient => marquage + ajout
         mark[j]=color, colonne[nbcoef++]=j;
//    4) tri des lignes par index de colonne
  for(i=0; i<n; i++)
    HeapSort(colonne + ligne[i-1] + 1 ,ligne[i] - ligne[i-1]);
//    nettoyage memoire
  delete  [] head_s;
  delete  [] next_p;
  delete  [] mark;
}
```

Au passage, nous avons utilisé la fonction `HeapSort` qui implémente un petit algorithme de tri, présenté dans [Knuth-1975], qui a la propriété d'ête toujours en $nlog_2n$ (*cf.* 4.6 page 52). Noter que l'étape de tri n'est pas absolument nécessaire, mais le fait d'avoir des lignes triées par indice de colonne permet d'optimiser l'accés à un coefficient de la matrice dans la structure

creuse, en utilisant une recherche dichotomique comme suit :

```
inline double*  MatriceMorseSymetrique::pij(int i,int j) const
 {
   assert(i<this->n && j< this->n);
   if (j > i ) {  int k=i; i=j;  j=k;}                    //    echange i et j
   int i0= ligne[i];
   int i1= ligne[i+1]-1;
   while (i0<=i1)                                            //    dichotomie
    {
      int im=(i0+i1)/2;
      if (j< colonne[im]) i1=im-1;
      else if (j> colonne[im]) i0=im+1;
      else return a+im;
    }
   return 0;                                          //    le coef n'existe pas
 }
```

Remarque : Si vous avez tout compris dans ces algorithmes, vous pouvez vous attaquer à la plupart des problèmes de programmation.

4.9.3 Le constructeur de la classe **SparseMatrix**

En modifiant légèrement l'algorithme précédent on obtient le constructeur de la classe `SparseMatrix` suivant :

```
template<class R>
template<class Mesh>
SparseMatrix<R>::SparseMatrix(Mesh & Th)
  :
  VirtualMatrice<R>(Th.nv),n(Th.nv),m(Th.nv), nbcoef(0),
  i(0),j(0),a(0)
  {
    const int nve = Mesh::Element::nv;
    int end=-1;
    int nn= Th.nt*nve;
    int mm= Th.nv;
    KN<int> head(mm);
    KN<int> next(nn);
    KN<int> mark(mm);
    int color=0;
    mark=color;

    head = end;
    for (int p=0;p<nn;++p)
      {
        int s= Th(p/nve,p%nve);
        next[p]=head[s];
        head[s]=p;
      }
    nbcoef =0;
    n=mm;
    m=mm;
```

```
  int kk=0;
  for (int step=0;step<2;++step)                                          //    2 etapes
                                              //    une pour le calcul du nombre de coef
                                                     //    l'autre pour la construction
    {
      for (int ii=0;ii<mm;++ii)
        {
          color++;
          int ki=nbcoef;
          for (int p=head[ii];p!=end;p=next[p])
            {
              int k=p/nve;
              for (int l=0;l<nve;++l)
                {
                  int jj= Th(k,l);
                  if( mark[jj] != color)  //    un nouveau sommet de l'ensemble
                    if (step==1)
                      {
                        i[nbcoef]=ii;
                        j[nbcoef]=jj;
                        a[nbcoef++]=R();
                      }
                    else
                      nbcoef++;
                  mark[jj]=color;                     //    on colorie le sommet j;
                }
            }
          int ki1=nbcoef;
          if(step==1)
            HeapSort(j+ki,ki1-ki);
        }

      if(step==0)
        {
          cout << " Allocation des tableaux  " << nbcoef << endl;
          i= new int[nbcoef];
          j= new int[nbcoef];
          a= new R[nbcoef];
          kk=nbcoef;
          nbcoef=0;
        }


    }
}
```

5 Différentiation automatique

Les dérivées d'une fonction décrite par son implémentation numérique peuvent être calculées automatiquement et exactement par ordinateur, en utilisant la différentiation automatique (DA). La fonctionnalité de DA est très utile dans les programmes qui calculent la sensibilité par rapport aux paramètres et, en particulier, dans les programmes d'optimisation et de design.

5.1 Le mode direct

L'idée de la méthode directe est de différencier chaque ligne du code qui définit la fonction. Les différentes méthodes de DA se distinguent essentiellement par l'implémentation de ce principe de base (cf. [**?**]).

L'exemple suivant illustre la mise en œuvre de cette idée simple.

Exemple : Considérons la fonction $J(u)$, donnée par l'expression analytique suivante :

$$J(u) \quad \text{avec} \quad \begin{aligned} x &= u - 1/u \\ y &= x + \log(u) \\ J &= x + y. \end{aligned}$$

Nous nous proposons de calculer automatiquement sa dérivée $J'(u)$ par rapport à la variable u, au point $u = 2.3$.

Méthode : Dans le programme de calcul de $J(u)$, chaque ligne de code sera précédée par son expression différentiée (avant et non après à cause des instructions du type $x = 2 * x + 1$) :

$$\begin{aligned} &dx = du + du/(u * u) \\ &\mathbf{x = u - 1/u} \\ &dy = dx + du/u \\ &\mathbf{y = x + \log(u)} \\ &dJ = dx + dy \\ &\mathbf{J = x + y} \end{aligned}$$

Ainsi avons nous associé à toute variable (par exemple x) une variable supplémentaire, sa différentielle (dx). La différentielle devient la dérivée seulement une fois qu'on a spécifié la variable de dérivation. La dérivée (dx/du) est obtenue en initialisant toutes les différentielles à zéro en début de programme sauf la différentielle de la variable de dérivation (ex du) que l'on initialise à 1.

La valeur de la dérivée $J'(u)|_{u=2.3}$ est donc obtenue en exécutant le programme ci-dessus avec les valeurs initiales suivantes : $u = 2.3, du = 1$ et $dx = dy = 0$.

Remarque 12. *Les langages C,C++ , FORTRAN... ont la notion de* constante. *Donc si l'on sait que, par exemple, $a = 2$ dans tous le programme et que a ne changera pas, on n'est pas obligé de lui associer une différentielle. Par exemple, la fonction C*

```
float mul(const int a, float u)
{  float x; x=a*u; return x;}
```

se dérive comme suit :

```
float dmul(const int a, float u, float du)
{  float x,dx; dx = a*du; x=a*u; return dx;}
```

Les structures de contrôle (boucles et tests) présentes dans le code de définition de la fonction seront traitées de manière similaire. En effet, une instruction de test de type *if* où a est prédéfini,

```
y = a;
if ( u>0) x = u;
  else    x = 2*u;
J=x+y;
```

peut être vue comme deux programmes distincts :
• le premier calcule

```
y=a; x=u; J=x+y;
```

et avec la DA, il doit retourner

```
dy=0; y=a; dx=du; x=u; dJ=dx+dy; J=x+y;
```

• le deuxième calcule

```
y=a; x=2*u; J=x+y;
```

et avec la DA, il doit retourner

```
dy=0; y=a; dx=2*du; x=2*u; dJ=dx+dy; J=x+y;
```

Les deux programmes sont réunis naturellement sous la forme d'un unique programme

```
dy=0; y=a;
if (u>0) {dx=du; x=u;}
    else {dx=2*du; x=2*u;}
dJ=dx+dy; J=x+y;
```

Le même traitement est appliqué à une structure de type boucle :

```
x=0;
for( int i=1; i<= 3; i++) x=x+i/u;
cout << x << endl;
```

qui, en fait, calcule

```
x=0; x=x+1/u; x=x+2/u; x=x+3/u; cout << x << endl;
```

Pour la DA, il va falloir calculer

```
dx=0;x=0;
dx=dx-du/(u*u); x=x+1/u;
dx=dx-2*du/(u*u); x=x+2/u;
dx=dx-3*du/(u*u); x=x+3/u;
cout << x <<'\t'<< dx << endl;
```

ce qui est réalisé simplement par l'instruction :

```
dx=0;x=0;
for( int i=1; i<= 3; i++)
    { dx=dx-i*du/(u*u); x=x+i/u;}
cout << x <<'\t'<< dx << endl;
```

Limitations :

• Si dans les exemples précédents la variable booléene qui sert de test dans l'instruction `if` et/ou les limites de variation du compteur de la boucle `for` dépendent de u, l'implémentation décrite plus haut n'est plus adaptée. Il faut remarquer que dans ces cas, la fonction définie par ce type de programme n'est plus différentiable par rapport à la variable u, mais est différentiable à droite et à gauche et les dérivées calculées comme ci-dessus sont justes.

Exercice 14. ‖ Ecrire le programme qui dérive une boucle `while`.

• Il existe des fonctions non-différentiables pour des valeurs particulières de la variable (par exemple, $\sqrt{u}$ pour $u = 0$). Dans ce cas, toute tentative de différentiation automatique pour ces valeurs conduit à des erreurs d'exécution du type *overflow* ou *NaN* (not a number).

5.2 Fonctions de plusieurs variables

La méthode de DA reste essentiellement la même quand la fonction dépend de plusieurs variables. Considérons l'application $(u_1, u_2) \to J(u_1, u_2)$ définie par le programme suivant :

$$y_1 = l_1(u_1, u_2) \quad y_2 = l_2(u_1, u_2, y_1) \quad J = l_3(u_1, u_2, y_1, y_2) \tag{121}$$

En utilisant la méthode de DA décrite précédemment, nous obtenons :

$$\begin{aligned}
&dy_1 = \partial_{u_1} l_1(u_1, u_2)dx_1 + \partial_{u_2} l_1(u_1, u_2)dx_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2}) \\
&dy_2 = \partial_{u_1} l_2 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_2 dy_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}) \\
&dJ = \partial_{u_1} l_3 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_3 dy_1 + \partial_{y_2} l_3 dy_2 \\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2}).
\end{aligned}$$

Pour obtenir dJ pour (u_1, u_2) donnés, il faut exécuter le programme deux fois : une première fois avec $dx_1 = 1, dx_2 = 0$, ensuite, une deuxième fois, avec $dx_1 = 0, dx_2 = 1$.

Une meilleure solution est de dupliquer les lignes $dy_i = \cdots$ et les évaluer successivement pour $dx_i = \delta_{ij}$. Le programme correspondant :

$$\begin{aligned}
&d1y_1 = \partial_{u_1} l_1(u_1, u_2)d1x_1 + \partial_{u_2} l_1(u_1, u_2)d1x_2 \\
&d2y_1 = \partial_{u_1} l_1(u_1, u_2)d2x_1 + \partial_{u_2} l_1(u_1, u_2)d2x_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2}) \\
&d1y_2 = \partial_{u_1} l_2 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_2 d1y_1 \\
&d2y_2 = \partial_{u_1} l_2 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_2 d2y_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}) \\
&d1J = \partial_{u_1} l_3 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_3 d1y_1 + \partial_{y_2} l_3 d1y_2 \\
&d2J = \partial_{u_1} l_3 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_3 d2y_1 + \partial_{y_2} l_3 d2y_2 \\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2})
\end{aligned}$$

sera exécuté pour les valeurs initiales : $d1x_1 = 1,\ d1x2 = 0,\ d2x_1 = 0,\ d2x_2 = 1$.

5.3 Une bibliothèque de classes pour le mode direct

Il existe plusieurs implémentations de la différentiation automatique, les plus connues étant **Adol-C, ADIFOR** et **Odyssee**. Leur utilisation implique une période d'apprentissage importante ce qui les rend peu accessibles aux programmeurs débutants. C'est la raison pour laquelle nous présentons ici une implémentation utilisant le mode direct qui est très simple d'utilisation, quoique moins performante que le mode inverse.

On utilise la technique de surcharge d'opérateurs (voir chapitre **??**). Toutefois, cette technique n'est efficace pour le calcul de dérivées partielles que si le nombre de variables de dérivation est inférieur à la cinquantaine. Pour une implémentation similaire en FORTRAN 90, voir Makinen [**?**].

5.4 Principe de programmation

Considérons la même fonction

$$J(u) \quad \text{avec} \quad \begin{array}{l} x = 2u(u+1) \\ y = x + \sin(u) \\ J = x * y. \end{array}$$

Par rapport à la méthode décrite précédemment, nous allons remplacer chaque variable par un tableau de dimension deux, qui va stocker la valeur de la variable et la valeur de sa différentielle.

Le programme modifié s'écrit :

```
float y[2],x[2],u[2];
                                              //    dx = 2 u du + 2 du (u+1)
      x[1] = 2 * u[0] * u[1] + 2 * u[1] * (u[0] + 1);
                                                         //    x = 2 u (u+1)
      x[0] = 2 * u[0] * (u[0] + 1);
      y[1] = x[1] + cos(u[0])*u[1];
      y[0] = x[0] + sin(u[0]);
      J[1] = x[1] * y[0] + x[0] * y[1];
                                                             //    J = x * y
      J[0] = x[0] * y[0];
```

L'étape suivante de l'implémentation (voir [?], chapitre 4) consiste à créer une classe C++ qui contient comme données membres les tableaux introduits plus haut. Les opérateurs d'algèbre linéaire classiques seront redéfinis à l'intérieur de la classe pour prendre en compte la structure particulière des données membres. Par exemple, les opérateurs d'addition et multiplication sont définis comme suit :

5.5 Implémentation comme bibliothèque C++

Tous les fichiers sont dans l'archive `http://www.ann.jussieu.fr/~hecht/ftp/MN406/FH/autodiff.tar.bz`.

Afin de rendre possible le calcul des dérivées partielles (N variables), l'implémentation C++ de la DA va utiliser la classe suivante :

Listing 3: *(la classe `ddouble`)*

```
  #include <iostream>
using namespace std;

struct ddouble {
 double val,dval;
 ddouble(double x,double dx): val(x),dval(dx) {}
 ddouble(double x): val(x),dval(0) {}

};

inline ostream &  operator<<(ostream & f,const  ddouble & a)
{ f << a.val << " ( d = "<< a.dval << ") " ; return f;}
```

```
inline ddouble operator+(const ddouble & a,const ddouble & b)
   { return ddouble(a.val+b.val,a.dval+b.dval);}
inline ddouble operator+(const double & a,const ddouble & b)
   { return ddouble(a+b.val,b.dval);}

inline ddouble operator*(const ddouble & a,const ddouble & b)
   { return ddouble(a.val*b.val,a.dval*b.val+a.val*b.dval);}
inline ddouble operator*(const double & a,const ddouble & b)
   { return ddouble(a*b.val,a*b.dval);}
inline ddouble operator/(const ddouble & a,const ddouble & b)
   { return ddouble(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
inline ddouble operator/(const double & a,const ddouble & b)
   { return ddouble(a/b.val,(-a*b.dval)/(b.val*b.val));}
inline ddouble operator-(const ddouble & a,const ddouble & b)
   { return ddouble(a.val-b.val,a.dval-b.dval);}

inline ddouble operator-(const ddouble & a)
   { return ddouble(-a.val,-a.dval);}
inline ddouble sin(const ddouble & a)
   { return ddouble(sin(a.val),a.dval*cos(a.val));}
inline ddouble cos(const ddouble & a)
   { return ddouble(cos(a.val),-a.dval*sin(a.val));}
inline ddouble exp(const ddouble & a)
   { return ddouble(exp(a.val),a.dval*exp(a.val));}
inline ddouble fabs(const ddouble & a)
   { return (a.val > 0) ? a : -a;}
inline  bool operator<(const ddouble & a ,const ddouble & b)
   { return a.val < b.val ;}
```

---

voila un petit exemple d'utilisation de ces classes

```
template<typename R> R f(R x)
{
   R y=(x*x+1.);
   return y*y;
}

int main()
{
        ddouble x(2,1);
        cout << f(2.0) << " x = 2.0,  (x*x+1)^2 " << endl;
        cout << f(ddouble(2,1)) << "2 (2x) (x*x+1)   " << endl;
        return 0;
}
```

Mais de faite l'utilisation des (templates) permet de faire une utilisation recursive.

```
#include <iostream>
using namespace std;

template<class R> struct Diff {
```

```
 R val,dval;
 Diff(R x,R dx): val(x),dval(dx) {}
 Diff(R x): val(x),dval(0) {}
};

template<class R>  ostream &  operator<<(ostream & f,const  Diff<R> & a)
{ f << a.val << " ( d = "<< a.dval << ") " ; return f;}
template<class R> Diff<R> operator+(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val+b.val,a.dval+b.dval);}
template<class R> Diff<R> operator+(const R & a,const Diff<R> & b)
   { return Diff<R>(a+b.val,b.dval);}

template<class R> Diff<R> operator*(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val*b.val,a.dval*b.val+a.val*b.dval);}
template<class R> Diff<R> operator*(const R & a,const Diff<R> & b)
   { return Diff<R>(a*b.val,a*b.dval);}
template<class R> Diff<R> operator/(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
template<class R> Diff<R> operator/(const R & a,const Diff<R> & b)
   { return Diff<R>(a/b.val,(-a*b.dval)/(b.val*b.val));}
template<class R> Diff<R> operator-(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val-b.val,a.dval-b.dval);}

template<class R> Diff<R> operator-(const Diff<R> & a)
   { return Diff<R>(-a.val,-a.dval);}
template<class R>Diff<R> sin(const Diff<R> & a)
   { return Diff<R>(sin(a.val),a.dval*cos(a.val));}
template<class R> Diff<R> cos(const Diff<R> & a)
   { return Diff<R>(cos(a.val),-a.dval*sin(a.val));}
template<class R> Diff<R> exp(const Diff<R> & a)
   { return Diff<R>(exp(a.val),a.dval*exp(a.val));}
template<class R> Diff<R> fabs(const Diff<R> & a)
   { return  (a.val > 0) ? a : -a;}
template<class R> bool operator<(const Diff<R> & a ,const Diff<R> & b)
   { return a.val < b.val ;}
template<class R> bool operator<(const Diff<R> & a ,const R & b)
   { return a.val < b ;}
```

Si l'on veut calculer l'a racine d'une equation $f(x) = y$

```
template<typename R> R f(R x)
{
   return x*x;
}
template<typename R> R Newtow(R y,R x)
{
                                                        //     solve: f(x) -y = 0
                                                    //     x -= (f(x)-y) / df(x);
   while (1) {
   Diff<R> fff=f(Diff<R>(x,1));
   R ff=fff.val;
   R dff=fff.dval;
   cout << ff << endl;
   x = x - (ff-y)/dff;
   if (fabs(ff-y) < 1e-10) break;
```

```
    }
    return x;
}
```

Maintenant, il est aussi possible de re-différencier automatiquement l'algorithme de Newtow. Pour cela ; il suffit d'ecrit

```
int main()
{
    typedef double R;
    cout << " -- newtow (2)  = " << Newtow(2.0,1.) << endl;
    Diff<R> y(2.,1) ,  x0(1.,0.);                                  //    donne
    Diff<R> xe(sqrt(2.), 0.5/sqrt(2.));                      //    solution exact
    cout << "\n -- x = Newton " << y << " ," << x0 << ")"<<  endl;
    Diff<R> x= Newtow(y,x0) ;
    cout << " x = " << x << " ==  " <<  xe << " = xe    "  << endl;
    return 0;
}
```

Les resultats sont

```
[guest-rocq-135177:~/work/coursCEA/autodiff] hecht% ./a.out
 -- newtow (2)  = 1
2.25
2.00694
2.00001
2
1.41421
 -- x = Newton 2 ( d = 1)  ,1 ( d = 0) )
1 ( d = 0)
2.25 ( d = 1.5)
2.00694 ( d = 1.02315)
2.00001 ( d = 1.00004)
2 ( d = 1)
 x = 1.41421 ( d = 0.353553)  ==  1.41421 ( d = 0.353553)  = xe
```

Références

[S. Bourne] S. Bourne le système unix, InterEdition, Paris.

[Kernighhan,Pike] B.W. Kernighan, R. Pike L'environnement de programmation UNIX, InterEdition, Paris.

[1] [Kernighhan, Richie]B.W. Kernighhan et D.M. Richie Le Langage C, Masson, Paris.

[J. Barton, Nackman-1994] J. Barton, L. Nackman *Scientific and Engineering, C++*, Addison-Wesley, 1994.

[Ciarlet-1978] P.G. Ciarlet , *The Finite Element Method*, North Holland. n and meshing. Applications to Finite Elements, Hermès, Paris, 1978.

[Ciarlet-1982] P. G. Ciarlet *Introduction à l'analyse numérique matricielle et à l'optimisation*, Masson ,Paris,1982.

[Ciarlet-1991] P.G. Ciarlet , Basic Error Estimates for Elliptic Problems, in Handbook of Numerical Analysis, vol II, Finite Element methods (Part 1), P.G. Ciarlet and J.L. Lions Eds, North Holland, 17-352, 1991.

[2] I. Danaila, F. hecht, O. Pironneau : *Simulation numérique en C++* Dunod, 2003.

[Dupin-1999] S. Dupin *Le langage C++*, Campus Press 1999.

[Frey, George-1999] P. J. Frey, P-L George *Maillages*, Hermes, Paris, 1999.

[George,Borouchaki-1997] P.L. George et H. Borouchaki , *Triangulation de Delaunay et maillage. Applications aux éléments finis*, Hermès, Paris, 1997. Also as P.L. George and H. Borouchaki , *Delaunay triangulation and meshing. Applications to Finite Elements*, Hermès, Paris, 1998.

[FreeFem++] F. Hecht, O. Pironneau, K. Othsuka FreeFem++ : Manual http://www.freefem.org/

[Hirsh-1988] C. Hirsch *Numerical computation of internal and external flows*, John Wiley & Sons, 1988.

[Koenig-1995] A. Koenig (ed.) : *Draft Proposed International Standard for Information Systems - Programming Language C++*, ATT report X3J16/95-087 (ark@research.att.com)., 1995

[Knuth-1975] D.E. Knuth , The Art of Computer Programming, 2nd ed., *Addison-Wesley*, Reading, Mass, 1975.

[Knuth-1998a] D.E. Knuth The Art of Computer Programming, Vol I : Fundamental algorithms, *Addison-Wesley*, Reading, Mass, 1998.

[Knuth-1998b] D.E. Knuth The Art of Computer Programming, Vol III : Sorting and Searching, *Addison-Wesley*, Reading, Mass, 1998.

[Lachand-Robert] T. Lachand-Robert, A. Perronnet (http://www.ann.jussieu.fr/courscpp/)

[Lascaux et Théodor] P. lascaux et R. Théodor Analyse numérique matricielle appliquée a l'art de l'ingénieur, Tome 2 Masson, 1987

[Löhner-2001] R. Löhner Applied CFD Techniques, Wiley, Chichester, England, 2001.

[Lucquin et Pironneau-1996] B. Lucquin, O. Pironneau *Introduction au calcul scientifique*, Masson 1996.

[Numerical Recipes-1992] W. H. Press, W. T. Vetterling, S. A. Teukolsky, B. P. Flannery : *Numerical Recipes : The Art of Scientific Computing*, Cambridge University Press, 1992.

[Raviart,Thomas-1983] P.A. Raviart et J.M. Thomas, *Introduction à l'analyse numérique des équations aux dérivées partielles*, Masson, Paris, 1983.

[Richtmyer et Morton-1667] R. D. Richtmyer, K. W. Morton : *Difference methods for initial-value problems*, John Wiley & Sons, 1967.

[Shapiro-1991] J. Shapiro *A C++ Toolkit*, Prentice Hall, 1991.

[Stroustrup-1997] B. Stroustrup *The C++ Programming Language*, Third Edition, Addison Wesley, 1997.

[Wirth-1975] N Wirth *Algorithms + Dat Structure = program*, Prentice-Hall, 1975.

[Aho et al -1975] A. V. Aho, R. Sethi, J. D. Ullman , Compilers, Principles, Techniques and Tools, Addison-Wesley, Hardcover, 1986.

[Lex Yacc-1992] J. R. Levine, T. Mason, D. Brown Lex & Yacc, O'Reilly & Associates, 1992.

[Campione et Walrath-1996] M. Campione and K. Walrath The Java Tutorial : Object-Oriented Programming for the Internet, Addison-Wesley, 1996.

Voir aussi Integrating Native Code and Java Programs. `http://java.sun.com/nav/read/Tutorial/native1.1/index.html`.

[Daconta-1996] C. Daconta Java for C/C++ Programmers, Wiley Computer Publishing, 1996.

[Casteyde-2003] Christian Casteyde Cours de C/C++ `http://casteyde.christian.free.fr/cpp/cours`

[3] C. Berge, *Théorie des graphes*, Dunod, 1970.