Résolution des équations intégrales de B. Després avec la librairie éléments finis MÉLINA++

Eric DARRIGRAND – Daniel MARTIN
IRMAR - Université Rennes 1

11 mai 2009

Résumé

Dans ce document, nous proposons un exemple de résolution des équations intégrales de B. Després par la librairie éléments-finis MÉLINA++. Les équations intégrales de B. Després font intervenir plusieurs opérateurs intégraux impliqués dans de nombreuses autres formulations intégrales. Cet exemple consistue donc une belle démonstration des larges possibilités de la librairie MÉLINA++ en matière de résolution d'équations intégrales. D'autre part, il se veut le plus pédagogique possible quant à l'utilisation de la librairie.

Introduction

Les équations intégrales sont largement utilisées dans la résolution des problèmes de propagation d'onde en milieu extérieur. Elles permettent de résoudre un problème 3D non borné (propagation d'onde autour d'un obstacle Ω^-) en se ramenant à un problème 2D borné (courant ou potentiel sur la surface Γ de l'obstacle Ω^-). En contre partie, le problème initial 3D, bien conditionné et faisant intervenir des opérateurs locaux devient un problème dont les opérateurs sont globaux et a priori mal conditionnés. Le gain en précision des méthodes d'équations intégrales est particulièrement remarquable en comparaison avec des méthodes volumiques lorsque peu d'efforts sont fournis quant à la gestion de la surface artificielle simulant l'infini. De nombreux travaux ont alors été menés pour promouvoir les équations intégrales. Le caractère global des opérateurs est compensé par l'utilisation de méthodes rapides. Et le caractère "mal conditionné" a motivé le développement des équations intégrales de B. Després : Par le biais de manipulations matricielles, nous pouvons écrire un jeu d'équations intégrales sous une forme favorable à une résolution basée sur un double gradient conjugué.

Ce document a pour but de proposer une résolution des équations intégrales de B. Després avec la librairie MÉLINA++ (voir [1]). Il se compose d'une section de présentation succincte des équations intégrales de B. Després, une section de définition des objets numériques utiles à la résolution via MÉLINA++, une documentation sur les objets MÉLINA++ impliqués et une section offrant des résultats obtenus ainsi.

Ce document n'a pas vocation à donner de détails mathématiques sur la mise en place des équations intégrales de B. Després. Le lecteur désireux d'en apprendre davantage sur les équations intégrales pourra par exemple s'intéresser à [2], [3], [4]. Des détails sur le développement et le traitement des équations intégrales de B. Després peuvent se trouver dans [5], [6], [7], [8] et [9]. En particulier, nous reproduisons dans ce document des travaux détaillés dans [9].

1 Equations intégrales de Després pour le problème de Helmholtz 3D

1.1 Formulation intégrale de B. Després

Le système d'équations intégrales introduit par B. Després dérive en fait de la minimisation d'une fonctionnelle quadratique. Mais ce système peut aussi être obtenu par manipulations matricielles après introduction d'une inconnue complémentaire qui joue le rôle de multiplicateur de Lagrange dans la formulation du problème de minimisation. Ici, nous récrivons le système d'équations intégrales tel que dérivé dans [7]).

Nous souhaitons résoudre l'équation de Helmholtz avec condition d'impédance sur le bord :

$$\left\{\begin{array}{ll} \Delta u + \kappa^2 u = 0 \ , & \text{dans } \Omega^+ \ , \\ \dfrac{\partial u}{\partial n}_{/\Gamma} + \imath\kappa u_{/\Gamma} = g \ , & \text{sur } \Gamma \ , \\ \lim\limits_{r\to+\infty} r\left(\dfrac{\partial u}{\partial r} - \imath\kappa u\right) = 0 \ , & \end{array}\right. \tag{1}$$

où Ω^- est un domaine régulier borné de $\mathbb{R}^3$, de frontière Γ, et $\Omega^+ = \mathbb{R}^3 \backslash \overline{\Omega^-}$. g est une donnée caractérisant l'onde incidente et κ le nombre d'onde. La normale unitaire n est extérieure. Une solution $\mathcal{C}^2(\overline{\Omega^+})$ de

$$\left\{ \begin{array}{ll} \Delta u + \kappa^2 u = 0 \ , & \text{dans } \Omega^+ \ , \\ \lim\limits_{r \to +\infty} r \left(\dfrac{\partial u}{\partial r} - \imath \kappa u \right) = 0 \ , & \end{array} \right.$$

est donnée par la représentation intégrale

$$\kappa u = M(\kappa u_{/\Gamma}) - L(\frac{\partial u}{\partial n}_{/\Gamma}) \text{ dans } \Omega^+ \ . \tag{2}$$

L et M sont les potentiels de simple et double couche définis par : $\forall x \notin \Gamma$

$$Lp(x) = \kappa \int_\Gamma G(x,y)\, p(y)\, d\gamma(y) \quad \text{et} \quad M\varphi(x) = \int_\Gamma \frac{\partial G}{\partial n_y}(x,y)\, \varphi(y)\, d\gamma(y)$$

avec $G(x,y) = \dfrac{e^{\imath\kappa|x-y|}}{4\pi|x-y|}$.

L (resp. M) est prolongeable pour $p \in H^{-1/2}(\Gamma)$ (resp. $q \in H^{1/2}(\Gamma)$) et à valeurs dans $H^1_{\rm loc}(\overline{\Omega^+})$.

En notant $q = \kappa u_{/\Gamma}$ et $p = \dfrac{\partial u}{\partial n}_{/\Gamma}$, on peut établir un premier jeu d'équations :

$$\left[\begin{array}{cc} D & -K' - \dfrac{I}{2} \\ -K + \dfrac{I}{2} & S \end{array} \right] \left[\begin{array}{c} q \\ p \end{array} \right] = \left[\begin{array}{c} 0 \\ 0 \end{array} \right] \tag{3}$$

où S, K, K' et D sont définis par : $\forall x \in \Gamma$

$$\begin{array}{ll} Sp(x) = \kappa \displaystyle\int_\Gamma G(x,y)\, p(y)\, d\gamma(y) \ , & Dq(x) = \dfrac{1}{\kappa} \displaystyle\int_\Gamma \frac{\partial^2 G}{\partial n_x \partial n_y}(x,y)\, q(y)\, d\gamma(y) \ , \\ Kq(x) = \displaystyle\int_\Gamma \frac{\partial G}{\partial n_y}(x,y)\, q(y)\, d\gamma(y) \ , & K'p(x) = \displaystyle\int_\Gamma \frac{\partial G}{\partial n_x}(x,y)\, p(y)\, d\gamma(y) \ . \end{array} \tag{4}$$

La singularité de D peut être traitée en remarquant que :

$$\begin{array}{rl} < Dq, q' >_{L^2(\Gamma)} = \dfrac{1}{\kappa} \displaystyle\int_\Gamma \int_\Gamma G(x,y)\, [& \kappa^2 q(y)\, \overline{q'(x)}\, (n_x \cdot n_y) \\ & - \overrightarrow{\rm rot}_\Gamma q(y) \cdot \overrightarrow{\rm rot}_\Gamma \overline{q'(x)} \ ]\, d\gamma(y)\, d\gamma(x) \ . \end{array} \tag{5}$$

Par la suite, les équations intégrales de B. Després sont basées sur :

- le découpage du noyau de Green en partie singulière et partie régulière :

$$G(x,y) = G_r(x,y) + \imath\, G_i(x,y) \ , \ \text{avec} \left\{ \begin{array}{l} G_r(x,y) = \dfrac{\cos(\kappa|x-y|)}{4\pi|x-y|} \ , \\ G_i(x,y) = \dfrac{\sin(\kappa|x-y|)}{4\pi|x-y|} \ . \end{array} \right. \tag{6}$$

  en appliquant la même décomposition aux opérateurs intégraux, $S = S_r + \imath S_i$, $K = K_r + \imath K_i$, ..., où S_r, S_i, ... sont réels.

- l'introduction d'une nouvelle inconnue $(\mu, \lambda) = \imath(q,p)$, ce qui permet de récrire le système (3) :

$$\mathbf{K} \left[\begin{array}{c} q \\ p \end{array} \right] + \mathbf{M} \left[\begin{array}{c} \mu \\ \lambda \end{array} \right] = \left[\begin{array}{c} 0 \\ 0 \end{array} \right] \ , \tag{7}$$

  où

$$\mathbf{K} = \left[\begin{array}{cc} D_r & -K_r' - \dfrac{I}{2} \\ -K_r + \dfrac{I}{2} & S_r \end{array} \right] \quad \text{et} \quad \mathbf{M} = \left[\begin{array}{cc} D_i & -K_i' \\ -K_i & S_i \end{array} \right] \ .$$

  En combinant cette dernière avec la condition au bord, on obtient :

$$\left[\begin{array}{c} q \\ p \end{array} \right] + \mathbf{M} \left[\begin{array}{c} q \\ p \end{array} \right] - \mathbf{K}^* \left[\begin{array}{c} \mu \\ \lambda \end{array} \right] = \tilde{g} \ . \tag{8}$$

  avec $\tilde{g} = \left[\begin{array}{c} -\imath g \\ g \end{array} \right]$.

- le fait que **M** soit positive, et dérive de l'opérateur de champ lointain : $\mathbf{M} = A_\infty^* A_\infty$ avec A_∞ défini par : pour $p, q \in L^2(\Gamma)$ et $\hat{s} \in S^2$

$$\left(A_\infty \begin{bmatrix} q \\ p \end{bmatrix}\right)(\hat{s}) = \frac{\kappa}{4\pi}\int_\Gamma e^{-\imath\kappa y.\hat{s}}.(p(y) + \imath(\hat{s}.n_y)q(y))d\gamma(y) \ . \tag{9}$$

Ainsi, en combinant cette succession de considérations (dont (7) et (8)), on obtient finalement :

$$\begin{cases} X + A_\infty^* A_\infty X - \mathbf{K}^* Y = \tilde{g} \ , \\ \mathbf{K}X + A_\infty^* A_\infty Y = 0 \ , \end{cases} \tag{10}$$

où $X = \begin{bmatrix} q \\ p \end{bmatrix}$ et $Y = \imath X = \begin{bmatrix} \mu \\ \lambda \end{bmatrix}$.

La théorie de la condition inf-sup permet d'établir l'existence et l'unicité de X et l'existence de Y. Afin d'assurer l'unicité de Y, B. Després a suggéré le système modifié (voir [7]) :

$$\begin{cases} (Id + \beta)X + A_\infty^* A_\infty X - \mathbf{K}^* Y = \tilde{g} - \imath\beta Y \ , \\ \mathbf{K}X + (\beta + A_\infty^* A_\infty)Y = \imath\beta X \ . \end{cases} \tag{11}$$

où β est un paramètre strictement positif. Ce β-système admet une unique solution (X, Y) satisfaisant $Y = \imath X$.

Nous étendons à présent la formulation au cas d'un problème d'impédance généralisé, avec la condition au bord $\frac{\partial u}{\partial n}_{/\Gamma} + \imath\kappa Z u_{/\Gamma} = f$ sur Γ, où Z est un opérateur d'impédance à partie réel positive. Soit R l'opérateur de réflection associé $R = \dfrac{Id - Z}{Id + Z}$, alors le même système d'équations (11) s'écrit avec

$$\tilde{g} = \begin{bmatrix} -\imath g \\ g \end{bmatrix} \qquad \text{et} \qquad g = (1 + R)f + R(-p + \imath q)$$

Notant $F = (1 + R)\begin{bmatrix} -\imath f \\ f \end{bmatrix}$ et $N_R = R\begin{bmatrix} 1 & \imath \\ \imath & -1 \end{bmatrix}$, (11) devient

$$\begin{cases} (Id + \beta)X + A_\infty^* A_\infty X - \mathbf{K}^* Y = N_R X + F - \imath\beta Y \ , \\ \mathbf{K}X + (\beta + A_\infty^* A_\infty)Y = \imath\beta X \ . \end{cases} \tag{12}$$

La condition $\mid R \mid \leq 1$ assure l'existence et l'unicité de la solution. Dans ce document, nous ne considérons que le cas où Z et R sont scalaires.

1.2 Discrétisation éléments finis et méthode de résolution

1.2.1 Discrétisation

On considère une discrétisation éléments-finis de Lagrange $\mathbb{P}_1$. On se donne une triangulation $\mathcal{T}_h$ et l'espace discret $V_h = \text{Vect}\{\varphi_i \ ; \ i = 1, N\}$ avec $\varphi_i, i \in \{1, ..., N\}$ les fonctions de base $\mathbb{P}_1$ associées à $\mathcal{T}_h$, et N le nombre de sommets du maillage éléments finis ainsi défini et noté Γ_h.

Notons aussi $\psi_i = \begin{bmatrix} \varphi_i \\ 0 \end{bmatrix}$ et $\psi_{i+N} = \begin{bmatrix} 0 \\ \varphi_i \end{bmatrix}$, pour $i \in \{1, ..., N\}$.

Il en résulte le système discret

$$\begin{bmatrix} \mathcal{D}_\beta + \mathcal{A} & -\mathcal{K}^* \\ \mathcal{K} & \mathcal{B}_\beta + \mathcal{A} \end{bmatrix}\begin{bmatrix} X_h \\ Y_h \end{bmatrix} = \begin{bmatrix} \mathcal{N}_R & -\imath\mathcal{B}_\beta \\ \imath\mathcal{B}_\beta & 0 \end{bmatrix}\begin{bmatrix} X_h \\ Y_h \end{bmatrix} + \begin{bmatrix} F_h \\ 0 \end{bmatrix} \tag{13}$$

avec $\mathcal{B}_\beta$, $\mathcal{D}_\beta$, $\mathcal{K}$, $\mathcal{K}^*$, $\mathcal{A}$, $\mathcal{N}_R$ les matrices $2N \times 2N$ et F_h le vecteur de taille $2N$ définis par :

$$\begin{array}{llll} (F_h)_j = < F, \psi_j >_{\mathbb{V}_h} \ , & & (\mathcal{B}_\beta)_{ji} = < \beta\psi_i, \psi_j >_{\mathbb{V}_h} \ , \\ (\mathcal{D}_\beta)_{ji} = < (1 + \beta)\psi_i, \psi_j >_{\mathbb{V}_h} \ , & & (\mathcal{N}_R)_{ji} = < N_R\psi_i, \psi_j >_{\mathbb{V}_h} \ , \\ (\mathcal{K})_{ji} = < \mathbf{K}\psi_i, \psi_j >_{\mathbb{V}_h} \ , & & (\mathcal{K}^*)_{ji} = < \mathbf{K}^*\psi_i, \psi_j >_{\mathbb{V}_h} \ , \\ (\mathcal{A})_{ji} = < A_\infty\psi_i, A_\infty\psi_j >_{L^2(\mathcal{S}^2)} & \text{ou} & (\mathcal{A})_{ji} = < \mathbf{M}\psi_i, \psi_j >_{\mathbb{V}_h} \ , \end{array} \tag{14}$$

où $\mathbb{V}_h = V_h \times V_h$. La condition inf-sup discrète confirme aussi l'existence et l'unicité de la solution.

1.2.2 Résolution du système discret

La résolution du système discret (13) se fait selon une méthode de Jacobi relaxée : On se donne α un paramètre de relaxation et $\begin{bmatrix} X_h^{(0)} \\ Y_h^{(0)} \end{bmatrix}$ un vecteur initial tel que $Y_h^{(0)} = \imath X_h^{(0)}$. Le n^{eme} itéré est donné par la relation :

$$\begin{bmatrix} X_h^{(n)} \\ Y_h^{(n)} \end{bmatrix} = \alpha \begin{bmatrix} \tilde{X}_h^{(n)} \\ \tilde{Y}_h^{(n)} \end{bmatrix} + (1-\alpha) \begin{bmatrix} X_h^{(n-1)} \\ Y_h^{(n-1)} \end{bmatrix} ,$$

où $\begin{bmatrix} \tilde{X}_h^{(n)} \\ \tilde{Y}_h^{(n)} \end{bmatrix}$ est solution de

$$\begin{bmatrix} \mathcal{D}_\beta + \mathcal{A} & -\mathcal{K}^* \\ \mathcal{K} & \mathcal{B}_\beta + \mathcal{A} \end{bmatrix} \begin{bmatrix} \tilde{X}_h^{(n)} \\ \tilde{Y}_h^{(n)} \end{bmatrix} = \begin{bmatrix} \mathcal{N}_R & -\imath\mathcal{B}_\beta \\ \imath\mathcal{B}_\beta & 0 \end{bmatrix} \begin{bmatrix} X_h^{(n-1)} \\ Y_h^{(n-1)} \end{bmatrix} + \begin{bmatrix} F_h \\ 0 \end{bmatrix} . \tag{15}$$

En posant

$$\begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} \mathcal{N}_R & -\imath\mathcal{B}_\beta \\ \imath\mathcal{B}_\beta & 0 \end{bmatrix} \begin{bmatrix} X_h^{(n-1)} \\ Y_h^{(n-1)} \end{bmatrix} + \begin{bmatrix} F_h \\ 0 \end{bmatrix} ,$$

il s'agit de résoudre le système suivant en $\begin{bmatrix} X \\ Y \end{bmatrix}$:

$$\begin{cases} X = (\mathcal{D}_\beta + \mathcal{A})^{-1}(V + \mathcal{K}^* Y) \\ (\mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}\mathcal{K}^* + \mathcal{B}_\beta + \mathcal{A})Y = W - \mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}V \end{cases} . \tag{16}$$

Pour résoudre l'équation en Y, on utilise un double gradient conjugué. La méthode du gradient conjugué appliquée à $(\mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}\mathcal{K}^* + \mathcal{B}_\beta + \mathcal{A})$ induit des multiplications par $(\mathcal{D}_\beta + \mathcal{A})^{-1}$ qui sont effectuées par application du gradient conjugué à $(\mathcal{D}_\beta + \mathcal{A})$.

Concernant la convergence des gradients conjugués, les matrices $(\mathcal{D}_\beta + \mathcal{A})$ et $(\mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}\mathcal{K}^* + \mathcal{B}_\beta + \mathcal{A})$ sont hermitiennes définies positives. La convergence de la méthode de Jacobi relaxée est assurée sous la condition $\mid R \mid < 1$.

1.2.3 Le cas $R = 0$

Dans le cas où $R = 0$, nous pouvons nous passer du niveau itératif de Jacobi. Pour cela, il suffit de récrire le système (10) ainsi :

$$\begin{cases} (Id - \beta)X + A_\infty^* A_\infty X - \mathbf{K}^* Y - \imath\beta Y = \tilde{g} , \\ (\mathbf{K} - \imath\beta)X + (\beta + A_\infty^* A_\infty)Y = 0 . \end{cases} \tag{17}$$

Après discrétisation, il s'agit alors de résoudre le système suivant en $\begin{bmatrix} X \\ Y \end{bmatrix}$ (au lieu de (13) résolu via (16)) :

$$\begin{cases} X = (\mathcal{D}_{-\beta} + \mathcal{A})^{-1}(F_h + (\mathcal{K}^* + \imath\mathcal{B}_\beta)Y) \\ ((\mathcal{K} - \imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta} + \mathcal{A})^{-1}(\mathcal{K}^* + \imath\mathcal{B}_\beta) + \mathcal{B}_\beta + \mathcal{A})Y = -(\mathcal{K} - \imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta} + \mathcal{A})^{-1}F_h \end{cases} . \tag{18}$$

Le système est résolu par un double gradient conjugué en prenant en considération le caractère hermitien de la matrice définie positive $((\mathcal{K} - \imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta} + \mathcal{A})^{-1}(\mathcal{K}^* + \imath\mathcal{B}_\beta) + \mathcal{B}_\beta + \mathcal{A})$ effectivement complexe.

1.2.4 Le cas de Dirichlet, $R = -1$

Le cas de la condition au bord de Dirichlet $u_{/\Gamma} = f$ sur Γ, est traité comme le cas général aboutissant au système d'équations (11), avec

$$\tilde{g} = \begin{bmatrix} -\imath g \\ g \end{bmatrix} \qquad \text{et} \qquad g = 2\imath\kappa f + R(-p + \imath q)$$

ce qui équivaut à (12) avec $F = 2\imath\kappa \begin{bmatrix} -\imath f \\ f \end{bmatrix}$

2 Définition des objets numériques et mise en œuvre de la résolution

2.1 Vecteurs et matrices

Il nous faut tout d'abord définir les inconnues et la discrétisation associée :

$$X = \begin{bmatrix} q \\ p \end{bmatrix} \text{ et } Y = \begin{bmatrix} \mu \\ \lambda \end{bmatrix} \quad \in \mathbb{C}^{2N \times 2N} \qquad \text{de type } \mathbb{P}_1 \text{ sur chaque composante.}$$

Ensuite, il nous faut définir quelques vecteurs :

$$f_h = (< f, \varphi_i >_{V_h})_{i=1,..,N} \in \mathbb{C}^N \quad ; \quad F_h = \begin{bmatrix} -\imath(1+R)f_h \\ (1+R)f_h \end{bmatrix} \quad ;$$

et matrices creuses $N \times N$:

$$I_h = (< \varphi_j, \varphi_i >_{V_h})_{i,j=1,..,N} .$$

Cela suffira à caractériser les matrices $2N \times 2N$:

$$\mathcal{B}_\beta = \begin{bmatrix} \beta I_h & 0 \\ 0 & \beta I_h \end{bmatrix} \quad , \quad \mathcal{D}_\beta = \begin{bmatrix} (\beta+1) I_h & 0 \\ 0 & (\beta+1) I_h \end{bmatrix} \quad , \quad \mathcal{N}_R = R \begin{bmatrix} I_h & \imath I_h \\ \imath I_h & -I_h \end{bmatrix} \quad .$$

Les matrices pleines $2N \times 2N$ sont toutes définies à partir des matrices $S_{h,r}$, $K_{h,r}$, ..., $K'_{h,i}$, $D_{h,i}$ de taille $N \times N$, selon les relations ci-dessous :

$$\mathcal{A} = \begin{bmatrix} D_{h,i} & -K'_{h,i} \\ -K_{h,i} & S_{h,i} \end{bmatrix} , \mathcal{K} = \begin{bmatrix} D_{h,r} & -K'_{h,r} - \frac{1}{2} I_h \\ -K_{h,r} + \frac{1}{2} I_h & S_{h,r} \end{bmatrix} , \mathcal{K}^* = \begin{bmatrix} D_{h,r} & -K'_{h,r} + \frac{1}{2} I_h \\ -K_{h,r} - \frac{1}{2} I_h & S_{h,r} \end{bmatrix} .$$

Les matrices de taille $N \times N$ sont définies selon les modèles ci-dessous utilisant les définitions (4, 5) :

$$S_{h,r} = (< S_r \varphi_j, \varphi_i >_{V_h})_{i,j=1,..,N} \quad \text{et} \quad D_{h,i} = (< D_i \varphi_j, \varphi_i >_{V_h})_{i,j=1,..,N} .$$

Ces dernières sont prévues dans la structure des opérateurs intégraux de surface de MÉLINA++.

2.2 Fonctions/méthodes propres à l'utilisateur

Nous introduisons ici les outils qui doivent être définis par l'utilisateur pour résoudre les équations intégrales de Després selon les méthodes itératives mentionnées. Ces outils consistent principalement à gérer la structure bloc des équations de Després à partir des opérateurs intégraux classiques connus de MÉLINA++. On n'utilisera pas ici la notion d'assemblage de la librairie car elle nous empêcherait d'exploiter au mieux les propriétés des blocs constituant le système global.

Remarque : Une particularité essentielle de ce travail sera développée plus tard : on souhaite résoudre un système linéaire en utilisant les solveurs de la librairie et sans vouloir assembler la matrice. Cela est rendu possible par la structure "template" des solveurs de MÉLINA++. Pour cela, il suffit de définir sa propre classe de matrice, de vecteur et sa propre opération de multiplication. En fonction du solveur, les différents outils à définir relativement à ces classes sont listés dans les commentaires-documentation du fichier `$src/iterative_solvers/IterativeSolver.h++`. Mais nous y reviendrons plus en détail dans la section 3.

Avec `Fh` = $\begin{bmatrix} \texttt{Fh1} \\ \texttt{Fh2} \end{bmatrix}$, `X` = $\begin{bmatrix} \texttt{X1} \\ \texttt{X2} \end{bmatrix}$, `Y` = $\begin{bmatrix} \texttt{Y1} \\ \texttt{Y2} \end{bmatrix} \in \mathbb{C}^{N+N}$, `f_h` donné par la structure `TermVector` de MÉLINA++ et `MultSr`, `MultKr`, ..., `MultKpi`, `MultDi` donnés par la multiplication et la structure `IEMTermMatrix` associées aux opérateurs intégraux de MÉLINA++, nous définissons ci-dessous les fonctions/méthodes "utilisateur".

Calcul de F_h

```
CalculFh
Fh1 = - i*(1+R)*f_h  ;   Fh2 = (1+R)*f_h ;
```

Multiplication par $\mathcal{A}$

```
Y = MultA(X)
Y1 = MultDi(X1) - MultKpi(X2)  ;   Y2 = - MultKi(X1) + MultSi(X2) ;
```

Multiplication par $\mathcal{K}$

```
Y = MultK(X)
Y1 = MultDr(X1) - MultKpr(X2) - 0.5*MultId(X2) ;
Y2 = - MultKr(X1) + 0.5*MultId(X1) + MultSr(X2) ;
```

Multiplication par $\mathcal{K}^*$

```
Y = MultKp(X)
Y1 = MultDr(X1) - MultKpr(X2) + 0.5*MultId(X2) ;
Y2 = - MultKr(X1) - 0.5*MultId(X1) + MultSr(X2) ;
```

Multiplication par $\mathcal{B}_\beta$

```
Y = MultBb(X)
Y1 = beta * MultId(X1)  ;   Y2 = beta * MultId(X2) ;
```

Multiplication par $\mathcal{D}_\beta$

```
Y = MultDb(X)
Y1 = (1+beta) * MultId(X1)  ;   Y2 = (1+beta) * MultId(X2) ;
```

Multiplication par $\mathcal{D}_{-\beta}$

```
Y = MultDmb(X)
Y1 = (1-beta) * MultId(X1)  ;   Y2 = (1-beta) * MultId(X2) ;
```

Multiplication par $\mathcal{N}_R$

```
Y = MultNR(X)
Xp1 = MultId(X1)  ;   Xp2 = MultId(X2) ;
Y1 = R * (Xp1 + i*Xp2)  ;   Y2 = R * (i*Xp1 - Xp2) ;
```

Multiplication par $(\mathcal{D}_\beta + \mathcal{A})$

```
Y = MultDbA(X)
Y = MultDb(X) + MultA(X) ;
```

Résolution du système $(\mathcal{D}_\beta + \mathcal{A})X = B$

```
X = Resol_DbA(B)
```
Résolution de type gradient conjugué faisant appel à la multiplication `MultDbA`.

Multiplication par $(\mathcal{D}_{-\beta} + \mathcal{A})$

```
Y = MultDmbA(X)
Y = MultDmb(X) + MultA(X) ;
```

Résolution du système $(\mathcal{D}_{-\beta} + \mathcal{A})X = B$

```
X = Resol_DmbA(B)
```
Résolution de type gradient conjugué faisant appel à la multiplication `MultDmbA`.

Multiplication par $(\mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}\mathcal{K}^* + \mathcal{B}_\beta + \mathcal{A})$

```
Y = MultBigMat(X)
Y = MultK( Resol_DbA( MultKp(X) ) ) + MultBb(X) + MultA(X) ;
```

Résolution du système $(\mathcal{K}(\mathcal{D}_\beta + \mathcal{A})^{-1}\mathcal{K}^* + \mathcal{B}_\beta + \mathcal{A})X = B$

```
X = Resol_BigM(B)
```
Résolution de type gradient conjugué faisant appel à la multiplication `MultBigMat`.

Multiplication par $((\mathcal{K} - \imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta} + \mathcal{A})^{-1}(\mathcal{K}^* + \imath\mathcal{B}_\beta) + \mathcal{B}_\beta + \mathcal{A})$

```
Y = MultBigMatR0(X)
Xp = Resol_DmbA( MultKp(X) + i*MultBb(X) )
Y = MultK(Xp) - i*MultBb(Xp) + MultBb(X) + MultA(X) ;
```

Résolution du système $((\mathcal{K} - \imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta} + \mathcal{A})^{-1}(\mathcal{K}^* + \imath\mathcal{B}_\beta) + \mathcal{B}_\beta + \mathcal{A})X = B$

```
X = Resol_BigMR0(B)
```
Résolution de type gradient conjugué faisant appel à la multiplication `MultBigMatR0`.

2.3 Résolution du système – méthode de Jacobi relaxée

2.3.1 Cas où $R \neq 0$.

L'algorithme de résolution du système (13) d'équations intégrales de B. Després est donné par :

```
(X,Y) = RelaxedJacobi(alpha)
(X,Y) <--- (X0,Y0)
Tant que ( n <= nbIterMax  ou  precision insuffisante ) faire :
     n <--- n+1
     V = MultNR(X) - i*MultBb(Y) + Fh
     W = i*MultBb(X) - MultK( Resol_DbA(V) )
     Yp = Resol_BigM(W)
     Xp = Resol_DbA( V + MultKp(Yp) )
     (X,Y) <--- alpha*(Xp,Yp) + (1-alpha)*(X,Y)
```

Remarque : En plus d'un critère d'arrêt usuel, on exploitera ici la différence entre Y et $\imath X$ afin de confirmer la précision de la méthode.

2.3.2 Cas où $R = 0$.

Dans le cas où $R = 0$, le niveau itératif de Jacobi disparaît et l'algorithme de résolution de (18) s'écrit :

```
(X,Y) = Resolution(R=0)
V = Resol_DmbA(Fh)
W = - MultK(V) + i*MultBb(V)
Y = Resol_BigMR0(W)
X = Resol_DmbA( Fh + MultKp(Y) + i*MultBb(Y) )
```

Remarque : La différence entre Y et $\imath X$ reste utile pour confirmer la précision de la méthode.

2.4 Structure du programme principal C++

La structure du programme principal est relativement simple. Celui-ci comprend les étapes suivantes :

- Initialisation des paramètres impliqués dans les équations intégrales de Després (nombre d'onde, coefficient d'impédance, coefficient β du β-système, paramètres relatifs aux solveurs : précision, nombre max d'itérations, paramètre de relaxation, ...).
- Définition de la fonction de Green
- Définition du domaine (maillage et domaine discret) et choix de l'élément fini.
- Déclaration et calcul des opérateurs et vecteurs MÉLINA++ impliqués (des `TermMatrix` et `TermVector`).
- Déclaration et calcul des matrices et vecteurs propres à la méthode de Després.
- Résolution du système $\longrightarrow$ méthode de Jacobi relaxée ou pas selon R.
- Ecriture des résultats.

3 Le programme en C++

Remarque : Dans ce qui suit, nous désignerons par `$melina++` le dossier contenant la librairie MÉLINA++. Il s'agira en général de `/usr/local/melina++` ou `/Library/Melina++`. De même, `$src` désignera le dossier `$melina++/src` et `$eid` désignera le dossier `$melina++/Examples/DespresIntEq`.

Comme nous l'avons annoncé plus haut, nous sommes tenus de définir nos propres classes de vecteurs et de matrices pour pouvoir résoudre les systèmes linéaires correspondants. Pour cela, nous suivons les conseils dictés dans le fichier `$src/iterative_solvers/IterativeSolver.h++`.

3.1 La classe vecteur

Les vecteurs avec lesquels nous souhaitons travailler sont en fait des paires de vecteurs et notre classe vecteur sera définie selon ce constat. En effet, par sa structure, la paire au sens c++ est un objet tout à fait adapté à nos besoins. Nous désignerons un élément de notre classe par "`vectorPair`"

Afin de pouvoir effectuer les opérations classiques sur ces `vectorPair`, nous devons redéfinir toutes ces opérations. Nous le faisons en exploitant au maximum le fait que ces opérations élémentaires existent déjà pour les "`TermVector`"

de la librairie MÉLINA++, objets à partir desquels sont construits nos `vectorPair`. Elles sont définies au travers de méthodes de la classe `vectorPair` et de fonctions externes à la classe.

Finalement, notre classe de vecteurs est définie ainsi :

```
class vectorPair : public pair<TermVector, TermVector>
{
public:
   // constructors
   vectorPair():pair<TermVector, TermVector>(){}
   vectorPair(const TermVector TV1, const TermVector TV2)
     :pair<TermVector, TermVector>(TV1,TV2){}
   vectorPair(const me_Type_t TVtype, const vectorPair& TVpair, const string& name)
     :pair<TermVector, TermVector>(TermVector(TVtype,TVpair.first,name)
                                   ,TermVector(TVtype,TVpair.second,name)){}
   //
   // Type of the components of the object
   me_Type_t t_type() const { return typeOfResult(first, second); }
   //
   // Below: The basic operations on vectors. Following the class TermVector
   //
   vectorPair& operator+=(const vectorPair& t)
   {
      first += t.first;
      second += t.second;
      return *this;
   }
   // Et bien d'autres dont -=, *=.
   // ---> Voir le fichier $eid/eid.c++
   //
   void zero()
   {
      first.zero();
      second.zero();
   }
   size_t size() const
   {
      return (first.size() + second.size());
   }
   // Et bien d'autres dont dotRC, dotC, norm2, norminfty.
   // ---> Voir le fichier $eid/eid.c++
   //
}; // end of class vectorPair.
```

Et les fonctions externes à la classe s'écrivent :

```
void add(const vectorPair& t1, vectorPair& t2)
{
   add(t1.first, t2.first);
   add(t1.second, t2.second);
}
void scale(const real_t scalar, const vectorPair& t1, vectorPair& t2)
{
   scale(scalar, t1.first, t2.first);
   scale(scalar, t1.second, t2.second);
}
void scale(const complex_t scalar, const vectorPair& t1, vectorPair& t2)
{
   scale(scalar, t1.first, t2.first);
   scale(scalar, t1.second, t2.second);
}
// Et bien d'autres dont scale_then_add, +, -, *, dotRC, dotC.
// ---> Voir le fichier $eid/eid.c++
```

3.2 Les classes matrices

Nous souhaitons pouvoir utiliser le gradient conjugué avec quatre matrices différentes. Deux d'entre elles sont très senblables : $\mathcal{D}_\beta + \mathcal{A}$ et $\mathcal{D}_{-\beta} + \mathcal{A}$. Elles diffèreront dans la programmation par la valeur d'un paramètre qui prendra, selon les cas, la valeur β ou $-\beta$.

Nous définissons alors une première classe matrice "`EIDAuxMatrix`" qui définira les matrices auxiliaires $\mathcal{D}_\beta + \mathcal{A}$ et $\mathcal{D}_{-\beta} + \mathcal{A}$. Cette classe contient les méthodes `MultA` et `MultDb/Dmb`. D'autre part, elle doit contenir la méthode `mat_Vec` qui définit le produit matrice-vecteur tel qu'il sera utilisé par le solveur "template" (il s'agit de `MultDbA`). Enfin, on y ajoute `adEIDResolCG`, une méthode de définition de la résolution `Resol_DbA/DmbA`.

La classe est définie ainsi :

```
class EIDAuxMatrix
{
protected:
   const TermMatrix* M_Si, *M_Ki, *M_Kpi, *M_Di, *M_Id;
   real_t beta1, epsil_i;
   number_t iterMax_i;
   //
public:
   // constructor
   EIDAuxMatrix(const TermMatrix& Si, const TermMatrix& Ki,
                const TermMatrix& Kpi, const TermMatrix& Di,
                const TermMatrix& Id, const real_t bet1,
                const real_t epsilon, const number_t maxIter )
   : M_Si(&Si), M_Ki(&Ki), M_Kpi(&Kpi), M_Di(&Di), M_Id(&Id),
     beta1(bet1), epsil_i(epsilon), iterMax_i(maxIter) {}
   //
   // destructor
   virtual ~EIDAuxMatrix(){M_Si = 0; M_Ki = 0; M_Kpi = 0; M_Di = 0;
                           M_Id = 0; beta1 = 0; epsil_i = 0; iterMax_i = 0 ;}
   //
   // Type of the EIDAuxMatrix (A+Db).
   me_Type_t t_type() const { return Real; }
   //
   vectorPair MultA(const vectorPair& X) const
   {
      TermVector Y1 = *M_Di * X.first - *M_Kpi * X.second;
      TermVector Y2 = - (*M_Ki * X.first) + *M_Si * X.second;
      return vectorPair(Y1,Y2);
   }
   //
   vectorPair MultDb(const vectorPair& X) const
   {
      TermVector Y1 = beta1 * *M_Id * X.first;
      TermVector Y2 = beta1 * *M_Id * X.second;
      return vectorPair(Y1,Y2);
   }
   //
   // Multiplication of (Db+A) by a vectorPair
   void mat_Vec(const vectorPair& X, vectorPair& Y) const
   {
      Y = MultA(X)+MultDb(X);
   }
   //
   // Resolution of a system the matrix of which is (Db+A).
   vectorPair adEIDResolCG(const vectorPair& B) const
   {
      vectorPair Bres(B.t_type(), B, "B");
      CgSolver CGp(epsil_i, iterMax_i);
      return CGp(*this,Bres);
   }
}; // end of class EIDAuxMatrix
```

Les deux autres matrices, $(\mathcal{K}(\mathcal{D}_\beta+\mathcal{A})^{-1}\mathcal{K}^*+\mathcal{B}_\beta+\mathcal{A})$ et $((\mathcal{K}-\imath\mathcal{B}_\beta)(\mathcal{D}_{-\beta}+\mathcal{A})^{-1}(\mathcal{K}^*+\imath\mathcal{B}_\beta)+\mathcal{B}_\beta+\mathcal{A})$, ont de nombreux points communs mais ne peuvent être définies au travers d'une seule classe. Nous créons alors une classe matrice "`EIDMatrix`" qui contiendra tous les éléments communs. Puis une classe par matrice sera créée comme fille de cette classe commune : "`EIDMatrixRnot0`" et "`EIDMatrixR0`" (on parle alors d'héritage). Ces dernières nécessitent l'appel de $\mathcal{D}_\beta+\mathcal{A}$ ou $\mathcal{D}_{-\beta}+\mathcal{A}$. On a donc choisi de définir `EIDMatrix` à partir de `EIDAuxMatrix` par héritage. De même que pour `EIDAuxMatrix`, elles vont contenir l'ensemble des méthodes requises pour la résolution du problème.

La classe `EIDMatrix` s'écrit alors :

```
class EIDMatrix : public EIDAuxMatrix
{
   const TermMatrix* M_Sr, *M_Kr, *M_Kpr, *M_Dr, *M_Id_2;
protected:
   const real_t beta, epsil_o;
   const number_t iterMax_o;
public:
   //
   // constructor
   EIDMatrix(const TermMatrix& Sr, const TermMatrix& Kr,
             const TermMatrix& Kpr, const TermMatrix& Dr,
             const TermMatrix& Si, const TermMatrix& Ki,
             const TermMatrix& Kpi, const TermMatrix& Di,
             const TermMatrix& Id, const TermMatrix& Id_2,
             const real_t bet, const real_t bet1,
             const real_t epsilono, const number_t maxItero,
             const real_t epsiloni, const number_t maxIteri  )
   : EIDAuxMatrix(Si,Ki,Kpi,Di,Id, bet1, epsiloni, maxIteri),
     M_Sr(&Sr), M_Kr(&Kr), M_Kpr(&Kpr), M_Dr(&Dr),
     M_Id_2(&Id_2), beta(bet), epsil_o(epsilono), iterMax_o(maxItero){}
   //
   vectorPair MultK(const vectorPair& X) const
   {
      TermVector Y1 = *M_Dr * X.first - *M_Kpr * X.second - *M_Id_2 * X.second;
      TermVector Y2 = - (*M_Kr * X.first) + *M_Id_2 * X.first + *M_Sr * X.second;
      return vectorPair(Y1,Y2);
   }
   //
   vectorPair MultKp(const vectorPair& X) const
   {
      TermVector Y1 = *M_Dr * X.first - *M_Kpr * X.second + *M_Id_2 * X.second;
      TermVector Y2 = - (*M_Kr * X.first)  - *M_Id_2 * X.first + *M_Sr * X.second;
      return vectorPair(Y1,Y2);
   }
   //
   vectorPair MultBb(const vectorPair& X) const
   {
      TermVector Y1 = beta * *M_Id * X.first;
      TermVector Y2 = beta * *M_Id * X.second;
      return vectorPair(Y1,Y2);
   }
}; // end of class EIDMatrix
```

La classe `EIDMatrixRnot0` s'écrit :

```
class EIDMatrixRnot0 : public EIDMatrix // K (Db+A)^{-1} Kp + Bb+A
{
   const real_t reflex;
public:
   // constructor
   EIDMatrixRnot0(const TermMatrix& Sr, const TermMatrix& Kr,
                  const TermMatrix& Kpr, const TermMatrix& Dr,
```

```
                const TermMatrix& Si, const TermMatrix& Ki,
                const TermMatrix& Kpi, const TermMatrix& Di,
                const TermMatrix& Id, const TermMatrix& Id_2,
                const real_t& reflexion, const real_t& bet, const real_t& bet1,
                const real_t& epsilono, const number_t maxItero,
                const real_t& epsiloni, const number_t maxIteri  )
   : EIDMatrix(Sr, Kr, Kpr, Dr, Si, Ki, Kpi, Di, Id, Id_2, bet, bet1
                            , epsilono, maxItero, epsiloni, maxIteri),
     reflex(reflexion){}
   //
   // used only by this kind of matrix
   vectorPair MultNR(const vectorPair& X) const
   {
      TermVector Xp1 = *M_Id * X.first;
      TermVector Xp2 = *M_Id * X.second;
      TermVector Y1 = reflex * ( Xp1 + complex_t(0.,1.)*Xp2 );
      TermVector Y2 = reflex * ( complex_t(0.,1.)*Xp1 - Xp2 );
      return vectorPair(Y1,Y2);
   }
   //
   // Multiplication of a EIDMatrixRnot0 by a vector
   void mat_Vec(const vectorPair& X, vectorPair& Y) const
   {
      Y = MultK( adEIDResolCG( MultKp(X) ) ) + MultBb(X) + MultA(X) ;
   }
   //
   // Resolution of a system the matrix of which is of class EIDMatrixRnot0.
   vectorPair EIDResolCG(vectorPair& B) const
   {
      vectorPair Bres(B.t_type(), B, "B");
      CgSolver CGp(epsil_o, iterMax_o);
      return CGp(*this,Bres);
   }
}; // end of class EIDMatrixRnot0
```

Enfin, la classe `EIDMatrixR0` est définie par :

```
class EIDMatrixR0 : public EIDMatrix // (K-i*Bb) (Dmb+A)^{-1} (Kp+i*Bb) + Bb+A
{
public:
   // constructor
   EIDMatrixR0(const TermMatrix& Sr, const TermMatrix& Kr,
               const TermMatrix& Kpr, const TermMatrix& Dr,
               const TermMatrix& Si, const TermMatrix& Ki,
               const TermMatrix& Kpi, const TermMatrix& Di,
               const TermMatrix& Id, const TermMatrix& Id_2,
               const real_t& bet, const real_t& bet1,
               const real_t& epsilono, const number_t maxItero,
               const real_t& epsiloni, const number_t maxIteri  )
   : EIDMatrix(Sr, Kr, Kpr, Dr, Si, Ki, Kpi, Di, Id, Id_2, bet, bet1
                            , epsilono, maxItero, epsiloni, maxIteri){}
   //
   // Type of the EIDMatrix ((K-i*Bb) (Dmb+A)^{-1} (Kp+i*Bb) + Bb + A).
   me_Type_t t_type() const { return Complex; }
   //
   // Multiplication of a EIDMatrix by a vector
   void mat_Vec(const vectorPair& X, vectorPair& Y) const
   {
   // a completer par le lecteur -- TP2 -- atelier Melina 2009
   }
   //
```

```
   // Resolution of a system the matrix of which is of class EIDMatrixR0.
   vectorPair EIDResolCG(const vectorPair& B) const
   {
   // a completer par le lecteur -- TP2 -- atelier Melina 2009
   }
}; // end of class EIDMatrixR0
```

3.3 Produit matrice-vecteur utilisateur

On peut désormais définir l'opérateur "`*`" pour toutes les classes impliquant le solveur itératif :

```
vectorPair operator*(const EIDAuxMatrix& MAT, const vectorPair& t1)
{
   vectorPair T(t1.t_type(), t1, "t1");
   MAT.mat_Vec(t1,T) ;
   return T;
}

vectorPair operator*(const EIDMatrixRnot0& MAT, const vectorPair& t1)
{
   vectorPair T(t1.t_type(), t1, "t1");
   MAT.mat_Vec(t1,T) ;
   return T;
}

vectorPair operator*(const EIDMatrixR0& MAT, const vectorPair& t1)
{
   vectorPair T(Complex, t1, "t1");
   MAT.mat_Vec(t1,T) ;
   return T;
}
```

3.4 RHS

La détermination du second membre $f_h = (< f, \varphi_i >_{V_h})_{i=1,..,N}$ requiert l'intervention d'une fonction f à définir par l'utilisateur. Cette fonction est impliquée dans le calcul d'un `FETermVector` ou d'un `NVTermVector` ou ... Ces derniers acceptent une "`me_function`" en argument. Nous devons alors définir une `me_function`, désignée ci-dessous "`me_incidentWave`" :

```
me_function me_incidentWave(incidentWaveImp, paL_incidentWave);
```

où "`incidentWaveImp`" désigne la fonction f programmée en C++ et `paL_incidentWave` désigne une liste de paramètres de type "`me_parameterList`" contenant tous les arguments nécessaires à f, autres que le point courant en lequel sera appliquée la fonction (exemple : le nombre d'onde, l'impédance, la direction de l'onde incidente, ...).

Le second membre sera alors créé à partir d'une commande du type :

```
NVTermVector id_incWave(Gamma, u_h, me_incidentWave, "id_incWave");
```

Enfin, voici la fonction `incidentWaveImp` :

```
complex_t incidentWaveImp(const me_point& x, me_parameterList* paL_incidentWave)
{
   const dimen_t d(3);
   //
   me_parameter pa_k = (*paL_incidentWave)("k");
   const real_t wavenumber = real(pa_k);
   const complex_t eye_k = wavenumber * complex_t(0.,1.);
   //
   me_parameter pa_Z = (*paL_incidentWave)("Z");
   const real_t impedance = real(pa_Z);
   //
   vector<real_t> incDir(d);
   me_parameter pa_incDir = (*paL_incidentWave)("IncDir0");  incDir[0] = real(pa_incDir);
   pa_incDir = (*paL_incidentWave)("IncDir1");  incDir[1] = real(pa_incDir);
```

```
   pa_incDir = (*paL_incidentWave)("IncDir2");  incDir[2] = real(pa_incDir);
   //
   real_t x_dot_incDir = inner_product(x.begin(), x.end(), incDir.begin(), 0.);
   complex_t Z_exp_ik_x_incD = impedance * exp(eye_k * x_dot_incDir);
   return Z_exp_ik_x_incD;
}
```

3.5 La fonction `main`

Le programme principal se décompose en plusieurs parties reprises en détail dans la sous-section 3.6. Après quelques modifications pour éviter d'alourdir inutilement le présent document, il s'écrit comme suit (l'original se trouve dans `$eid/eid.c++`):

```
int main ()
{
   init();   // on peut mettre "init()" ou "init(fr)" ; par defaut, c'est "en".
#ifdef ME_DEBUG_ON
logon();
#endif
   //
   // ----------------------------------------------------------------
   // Read some datas related to Despres's integral formulation:
   //
   real_t wavenumber, epsilon_jac, epsilon_o, epsilon_i, reflex, alpha, beta;
   number_t nbIterJacMax, nbIterGCiMax, nbIterGCoMax, nbSphereSub;
   vector<real_t> incDirec(3);
   //
   string dataname("/Users/edarrigr/MyWork/Melina/Examples/DespresIntEq/data.txt")
            , outputname, comment_string;
   //
   ifstream data_file;
   data_file.open(dataname.c_str(), ios::in);
   // Read parameters followed by an ignored comment-string
   data_file>> outputname >> comment_string;
   // On lit de meme : wavenumber, nbSphereSub, incDirec[0], incDirec[1], incDirec[2]
   //     , reflex, beta, alpha, epsilon_jac, nbIterJacMax, epsilon_o, nbIterGCoMax
   //     , epsilon_i, nbIterGCiMax, number_t nb_given_rule
   //     , et les given-rules mises dans IEMParamList, un me_parameterList.
   //
   tpl_normalize(incDirec.begin(),incDirec.end(), 0.);
   real_t minus1_wavenumber = - 1. / wavenumber;
   real_t betap1 = 1.+beta;
   real_t betam1 = 1.-beta;
   real_t eps_machine(theEpsilon_);
   //
   thePrintStream_ <<endl<<"   ****************************    ";
   thePrintStream_ <<endl<<endl<<"   INTEGRAL EQUATIONS OF DESPRES    ";
   thePrintStream_ <<endl<<"   ****************************    ";
   thePrintStream_ <<endl<<"   wavenumber: "
                   <<wavenumber<<"   parameter of reflection: "<<reflex;
   //  eventuellement : imprimer tous les parametres.
   //
   // ----------------------------------------------------------------
   // Define the Green function.
   //
   me_parameter pa_wavenb(wavenumber,"k");
   me_parameterList paL_gf(pa_wavenb);
   GreenFunctionHelmholtz3d GF(paL_gf);
   //
   // ----------------------------------------------------------------
   // Define the discretization.
   //
```

```
Mesh sph_mesh=SubdivisionSphere(8,nbSphereSub,1,1,1.,"sphere");
//sph_mesh.print();
GeomDomain& Gamma=sph_mesh.domain("Sigma_1");
//Gamma.setVerboseLevel(vb);  //Gamma.print();
FESpace W_h(Gamma);   //W_h.print(10);
FEUnknown u_h("u_h",W_h);
//
// ---------------------------------------------------------------
// Define the vector/matrices involved in the Despres's system.
//
// Define Terms
//
TermMatrix S_r, K_r, Kp_r, D_r, S_i, K_i, Kp_i, D_i, Id_2;
FETermMatrix Id(Gamma, u_h, u_h,"MassFE");
vectorPair Fh;
//
IEMTermMatrix S(Gamma, u_h, u_h, id(GF), IEMParamList,"idGF_IEM");
IEMTermMatrix K(Gamma, u_h, u_h, ngrady(GF), IEMParamList,"nGradyGF_IEM");
IEMTermMatrix Kp(a faire par le lecteur -- TP2 -- atelier Melina 2009);
IEMTermMatrix Dnn(Gamma, nx(u_h), nx(u_h), id(GF), IEMParamList,"idGF_nxnyIEM");
IEMTermMatrix Dcurl(a faire par le lecteur -- TP2 -- atelier Melina 2009);
TermVector f_h;
//
// Create parameterList for rhs-functions
me_parameterList paL_incidentWave(pa_wavenb);
me_parameter pa_incD0(incDirec[0],"IncDir0"), pa_incD1(incDirec[1],"IncDir1")
           , pa_incD2(incDirec[2],"IncDir2");
paL_incidentWave << pa_incD0 << pa_incD1 << pa_incD2 ;
//
real_t OnePlusReflex = reflex + 1. ;
complex_t eye_k = wavenumber * complex_t(0.,1.) ;
if (OnePlusReflex<eps_machine)
{
   // Dirichlet case ---> a faire par le lecteur -- TP2 -- atelier Melina 2009.
}
else
{
   real_t impedance = (1. - reflex) / OnePlusReflex ;
   me_parameter pa_R(reflex,"R"), pa_Z(impedance,"Z");
   paL_incidentWave << pa_R << pa_Z ;
   // Define the rhs function and rhs-vectors.
   me_function me_incidentWave(incidentWaveImp, paL_incidentWave);
   me_function me_grad_incidentWave(grad_incidentWaveImp, paL_incidentWave);
   //
   NVTermVector dn_incWave(Gamma, u_h, ndot(me_grad_incidentWave), "dn_incWave");
   NVTermVector id_incWave(Gamma, u_h, me_incidentWave, "id_incWave");
   //
   // Compute Terms
   Gamma.computation(0);
   //
   f_h = (OnePlusReflex * eye_k) * (Id * (dn_incWave + id_incWave));
   // f_h.print(10);
   // Id.print(10);
}
Fh = vectorPair(- complex_t(0.,1.) * f_h , f_h );
S *= wavenumber ;
Dnn *= wavenumber;
Dcurl *= (minus1_wavenumber) ;
Dnn += Dcurl;
//
// Define matrices and vectors of the system from the terms
D_r = real(Dnn); D_i = imag(Dnn);
```

```
Kp_r = real(Kp); Kp_i = imag(Kp);
K_r = real(K); K_i = imag(K);
S_r = real(S); S_i = imag(S);
//
Id_2 = 0.5 * Id;
//
// ------------------------------------------------------------------
// Numerical resolution of Despres's system
//
vectorPair X(Fh), Y(Fh), Y_iX;
real_t norm_Y_iX, norm_X ;
if (abs(reflex)<eps_machine)
{  // cas ou le systeme ne necessite pas du niveau iteratif de Jacobi.
   //
   // Define the bock-matrices EIDAuxMatrix and EIDMatrixR0 - case R=0.
   EIDAuxMatrix DmbA(a faire par le lecteur -- TP2 -- atelier Melina 2009);
   // DmbA = Dmb + A
   EIDMatrixR0 MBM0(a faire par le lecteur -- TP2 -- atelier Melina 2009);
   // MBM0 = (K-i*Bb) (Dmb+A)^{-1} (Kp+i*Bb) + Bb + A
   //
   vectorPair V(Fh), W(Fh);
   V.zero(); W.zero();
   //
   // Resolution a ecrire par le lecteur -- TP2 -- atelier Melina 2009
   //
   cout<<endl<<"Accuracy estimation -- norm(Y-iX)/norm(X) = "<<norm_Y_iX;
   thePrintStream_<<endl<<"Accuracy estimation -- norm(Y-iX)/norm(X) = "<<norm_Y_iX;
}
else
{  // cas ou le systeme necessite le niveau iteratif de Jacobi.
   //
   // Define the bock-matrices EIDAuxMatrix and EIDMatrixRnot0 - case R=/=0.
   EIDAuxMatrix DbA(S_i, K_i, Kp_i , D_i, Id, betap1, epsilon_i, nbIterGCiMax);
   // DbA = Db + A
   EIDMatrixRnot0 MBM(S_r, K_r, Kp_r , D_r, S_i, K_i, Kp_i , D_i, Id, Id_2,
                      reflex, beta, betap1, epsilon_o, nbIterGCoMax,
                      epsilon_i, nbIterGCiMax);
   // MBM = K (Db+A)^{-1} Kp + Bb + A
   //
   number_t n(0);
   real_t norm_Xn_Xo, current_accuracy;
   vectorPair V(Fh), W(Fh), Yp(Fh), Xp(Fh), Xnew_Xprevious(Fh), Ynew_Yprevious(Fh);
   V.zero(); W.zero(); Xp.zero(); Yp.zero(); Xnew_Xprevious.zero(); Ynew_Yprevious.zero();
   X.zero(); Y.zero();
   do {
      n++;
      V = MBM.MultNR(X) - complex_t(0.,1.) * MBM.MultBb(Y) + Fh;
      W = complex_t(0.,1.) * MBM.MultBb(X) - MBM.MultK( DbA.adEIDResolCG(V) );
      Yp = MBM.EIDResolCG(W);
      Xp = DbA.adEIDResolCG(V+MBM.MultKp(Yp));
      //
      Xnew_Xprevious = alpha * (Xp - X);
      Ynew_Yprevious = alpha * (Yp - Y);
      norm_Xn_Xo = Xnew_Xprevious.norm2();
      //
      X += Xnew_Xprevious ;
      Y += Ynew_Yprevious ;
      // X = alpha * Xp + alpha_1 * X;   // with alpha_1 = 1 - alpha;
      // Y = alpha * Yp + alpha_1 * Y;
      norm_X = X.norm2();
      //
      Y_iX = Y - complex_t(0.,1.) * X;
```

```
        norm_Y_iX = Y_iX.norm2() /norm_X;
        //
        current_accuracy = norm_Xn_Xo / norm_X ;
        cout<<endl<<"Jacobi method iteration "<<n
                  <<" -- norm(Y-iX)/norm(X) = "<<norm_Y_iX;
        cout<<endl<<"Jacobi method iteration "<<n
                  <<" -- norm(Xnew-Xprevious)/norm(X) = "<<current_accuracy;
        thePrintStream_<<endl<<"Jacobi method iteration "<<n
                       <<" -- norm(Y-iX)/norm(X) = "<<norm_Y_iX;
        thePrintStream_<<endl<<"Jacobi method iteration "<<n
                       <<" -- norm(Xnew-Xprevious)/norm(X) = "<<current_accuracy;
       }
     while ((current_accuracy > epsilon_jac) && (n<nbIterJacMax));
     cout<<endl<<"Jacobi method used "<<n<<" iterations.";
     thePrintStream_<<endl<<"Jacobi method used "<<n<<" iterations.";
  }
  //
  ofstream result_file;
  result_file.open(outputname.c_str(), ios::out);
  X.first.output(result_file);
  X.second.output(result_file);
  Y.first.output(result_file);
  Y.second.output(result_file);
  //
  Cputime("Total");
}
```

3.6 Détails sur la fonction `main`

Nous commentons ici de manière détaillée les différentes parties de `main`.

3.6.1 Préambule

Tout d'abord, vous pouvez choisir que MÉLINA++ vous parle en Français ou en Anglais en écrivant "`init(fr)`" ou "`init()`" (= "`init(en)`"). Cependant, le compilateur C++ vous parlera toujours dans la même langue ;-).

MÉLINA++ offre un ensemble de variables et types prédéfinis que l'utilisateur est encouragé à utiliser. Ces objets sont visibles principalement dans les fichiers `me_config.h++` et `me_global_scope_data.h++` dans le dossier `$src/headers`.

3.6.2 Initialisation des paramètres

Nous avons choisi de lire les données relatives au problème et aux spécificités des équations intégrales de B. Després dans un fichier dont le nom est entré en dur dans la fonction `main`. Il suffit de définir ce fichier comme un "`ifstream`".

3.6.3 Fonction de Green

Dans notre cas, la fonction de Green est créée par le constructeur de la classe `GreenFunctionHelmholtz3d` faisant appel au constructeur de la classe `GreenFunction`. Ces deux constructeurs sont visibles dans les dossiers `$src/math_utils/` et `$src/math_utils/green_functions`. Le premier d'entre eux ne requiert qu'un argument : un `me_parameterList` (ou liste de paramètres) qui contient tous les éléments nécessaires à notre fonction de Green. Dans notre cas, il ne contient que le nombre d'onde. Les éléments possibles de ce `me_parameterList` sont listés dans les commentaires associés à notre fonction de Green dans `$src/math_utils/GreenFunction.h++`. On y découvre aussi la liste des fonctions de Green existantes.

Tout utilisateur peut intégrer sa propre fonction de Green. Il doit alors la définir comme celles déjà existantes : Ecrire un couple de fichiers h++/c++ correspondants dans le dossier `$src/math_utils/green_functions/`, contenant la définition et la programmation de cette fonction et dérivées successives si nécessaire.

Remarque : La définition des fonctions de Green est basée sur une stratégie adoptée pour le traitement des singularités. Lorsque la fonction de Green est singulière, le traitement des singularités utilise, aujourd'hui, des techniques de changements de variables proposés par Jean Gay, ingénieur au CEA, et utilisés dans de nombreuses thèses soutenues à Bordeaux. Ces techniques sont détaillées dans `$melina++/doc/biem_singularities/iem++.pdf`. Nous invitons le lecteur à s'y référer dans le cas où il souhaiterait définir ses propres fonctions de Green.

3.6.4 Maillage et domaine

Quelques maillages sont déjà disponibles dont celui défini par le constructeur de la classe `SubdivisionSphere` dont la documentation se trouve dans `$src/geometry/simple_meshes/SubdivisionSphere.h++`.

La définition du domaine se fait par simple référence au domaine du maillage qui nous intéresse. Dans notre cas, ce domaine s'appelle `"Sigma_1"`. Cette information se trouve dans `$src/geometry/subdivision/Boundary.c++` et n'est pas accessible de manière immédiate. Le renseignement de cette information est en cours d'évolution et deviendra plus consistant dans un futur proche.

3.6.5 Espace élément-fini et inconnue

L'espace élément-fini est créé par le constructeur `FESpace` défini dans le fichier `$src/space/FESpace.h++`. Les éléments-finis et degrés disponibles sont listés (`enum`) dans le fichier `$src/space/FEinterpolation.h++`.

On associe ensuite une inconnue à l'espace choisi par la fonction `FEUnknown`.

3.6.6 Définition d'un "`FETermMatrix`" ou d'un "`FE/NVTermVector`"

Le calcul des matrices et vecteurs relatifs à notre problème se fait par l'appel de la commande "`computation`" associée au domaine concerné. Mais avant cela, il est indispensable de définir tous les `FETermMatrix`, `IEMTermMatrix`, `FETermVector`, `NVTermVector` dont on aura besoin. Les `FETermMatrix` sont créés selon les constructeurs fournis dans `$src/FE_integrals/FETermMatrix.h++`. La liste des opérateurs applicables à l'inconnue est donnée dans le fichier `$src/operators/DifferentialOperator.h++`.

Concernant les `FETermVector` et `NVTermVector`, les mêmes informations sont disponibles dans les fichiers `$src/FE_integrals/FETermVector.h++` et `$src/nodalvalue/NVTermVector.h++`.

Ces termes peuvent nécessiter l'appel d'une function à laquelle on appliquera éventuellement un opérateur différentiel de la liste fournie dans `$src/operators/FunctionOperator.h++` (exemple : lors de la définition du second membre).

3.6.7 Définition d'un "`IEMTermMatrix`"

Le calcul des matrices relatives aux opérateurs intégraux nécessite le traitement de la singularité du noyau de Green associé. La technique utilisée par MÉLINA++ est définie dans `$melina++/doc/biem_singularities/iem++.pdf`. Cette technique fait intervenir un grand nombre de formules de quadrature dont le choix par défaut peut convenir ou ne pas convenir à l'utilisateur. La classe `IEMTermMatrix` contient alors plusieurs constructeurs offrant la possibilité de modifier ou non certaines ou toutes les formules de quadrature. Ces constructeurs, visibles et commentés dans le fichier `$src/IEM_integrals/IEMTermMatrix.h++`, sont semblables à ceux de la classe `FETermMatrix`. La différence provient de l'implication d'une fonction de Green et d'une liste de paramètres relative au traitement de ces singularités.

Les listes des opérateurs qui peuvent être associés à l'inconnue et à la fonction de Green sont données dans les fichiers `$src/operators/DifferentialOperator.h++` et `$src/operators/KernelOperator.h++`.

Pour modifier les formules de quadrature relatives au traitement des singularités, il suffit d'injecter les informations relatives dans un `me_parameterList` dénommé "`IEMParamList` dans l'exemple du fichier `$eid/eid.c++` et d'utiliser un constructeur de la classe `IEMTermMatrix` permettant la considération de cette liste de paramètres. Pour cela, l'utilisateur associera un numéro et degré pour chaque formule de quadrature dont il souhaite contrôler le choix. L'utilisateur devra d'abord prendre connaissance de :

- l'exemple `$eid/eid.c++`.
- la liste des noms associés aux formules de quadrature impliquées dans le traitement de singularité. Cela est visible dans le commentaire d'entête du fichier `$src/IEM_integrals/IEMTermMatrix.h++`.
- la liste des formules de quadratures disponibles et des degrés associés disponibles. Cela est visible dans le fichier `$src/elements/Quadrature.h++`.

3.6.8 Utilisation des objets éléments finis MÉLINA++

La commande "`computation`" provoque le calcul de tous les `FETermMatrix`, `IEMTermMatrix`, ainsi que les `FETermVector`, `NVTermVector` précédemment définis. Par la suite, il nous est alors possible de définir tous les `TermMatrix` et `TermVector` requis pour notre méthode et de lancer la résolution. Pour toute information sur les solveurs itératifs, le lecteur pourra se référer aux fichiers du dossier `$src/iterative_solvers/`.

3.6.9 Ecriture et exploitation du résultat

L'écriture du résultat se fait pour l'instant à l'aide d'une fonction `output` définie dans la classe `TermVector`. Le lecteur est dispensé du post-traitement des résultats qui sont exploités pour obtenir les jolies courbes de la section 4 où la solution de référence est donnée par les séries de Mie.

4 Résultats numériques

Les résultats numériques de cette section ont été obtenus sur un MacBook Pro 2.6 GHz Intel Core 2 Duo par utilisation du code `$eid/OptimVersion/eid_optim.c++`. Dans toute la section, $S1$, ..., $S5$ désigneront les maillages de sphères obtenus par subdivisions successives d'un maillage de la sphère $S0$ constitué de 8 triangles. k désignera le nombre d'onde, et R le coefficient de réflexion, égal à 0 dans le cas d'une condition de Robin, 1 dans le cas d'une condition de Neumann, et -1 dans le cas d'une condition de Dirichlet. Nous considérons systématiquement le cas d'une onde incidente plane de direction $(0, 0, -1)$. Les critères d'arrêt sont choisis égaux à 10^{-4} pour les gradients conjugués, à 10^{-2} (resp. 10^{-3}) pour la méthode de Jacobi dans le cas de Dirichlet (resp. de Neumann). Les figures montrent la SER en fonction de l'angle de rayonnement (écho radar à l'infinie, en fonction de la direction d'observation).

4.1 Cas $k = 2$

Nous présentons ici des résultats de convergence obtenus pour le cas $k = 2$. Le maillage $S3$ correspond à un maillage dont le pas h approche le dixième de la longueur d'onde, $\lambda/10$. Dans les figures 1, les solutions obtenues numériquement selon les différents maillages considérés sont comparées à la solution donnée par les séries de Mie. Dans le tableau 1, nous utilisons les notations suivantes :

- ⋆ nbCGi = nombre d'itérations du gradient conjugué interne à convergence.
- ⋆ nbCGo = nombre d'itérations du gradient conjugué externe à convergence.
- ⋆ nbJ = nombre d'itérations de la méthode de Jacobi à convergence.
- ⋆ Tps = Temps CPU en seconde.
- ⋆ Mem = Mémoire requise en Mo.
- ⋆ ErrRel = Erreur relative en norme 2 commise sur l'amplitude de diffusion.
- ⋆ Y-iX = $\|Y - \imath X\|_2 \,/\, \|X\|_2$.

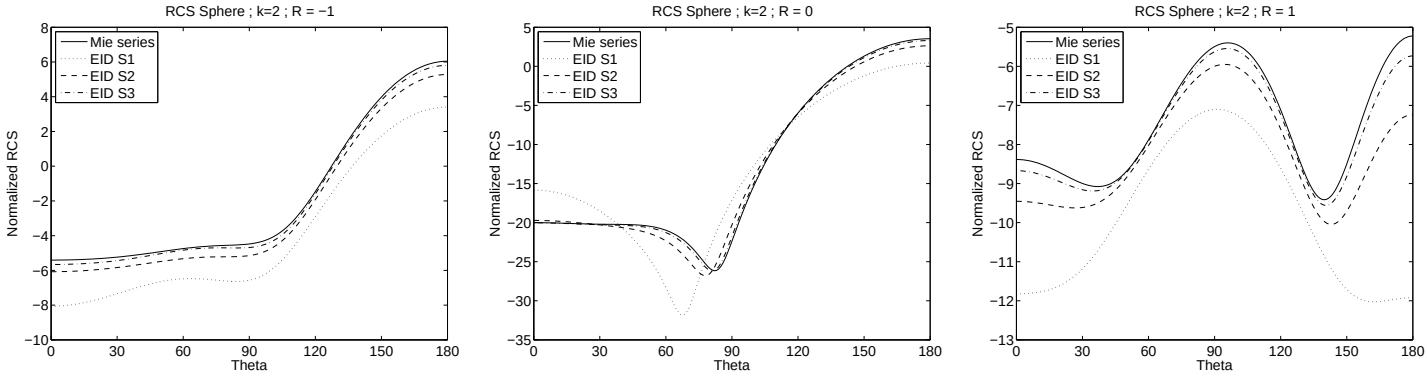

FIG. 1 – Cas où $k = 2$ et $R = -1\ ,\ 0\ ,\ 1$.

4.2 Cas $k = 4$ et $k = 8$

Nous initions ici la montée en fréquence en multipliant successivement par 2 le nombre d'onde, ce qui multiplie par 4 la taille du maillage correspondant et par 16 le coût de résolution. On peut en effet constater dans le tableau 2 la dépendance en k^4 du coût de résolution, motivant fortement l'introduction de méthodes rapides dans la librairie. Les résultats rappelés pour le nombre d'onde $k = 2$ sont ceux obtenus avec le maillage $S3$.

TAB. 1 – Coûts de résolution – cas $k = 2$

Cas	nbCGi	nbCGo	nbJ	Tps (s)	Mem (Mo)	ErrRel	Y-iX
$R = -1$, $S1$	8	12	98	4.57	2.4	0.2589	0.047273
$R = -1$, $S2$	11	14	26	9.65	2.6	0.0768	0.0172832
$R = -1$, $S3$	13	24	18	107.88	6.8	0.0229	0.0118533
$R = 0$, $S1$	9	11	–	0.92	2.4	0.3043	0.052717
$R = 0$, $S2$	12	14	–	7.12	2.6	0.0893	0.015145
$R = 0$, $S3$	14	27	–	86.64	6.8	0.0231	0.0056842
$R = 1$, $S1$	8	11	19	1.6	2.4	0.3136	0.0590852
$R = 1$, $S2$	12	15	19	8.93	2.6	0.1089	0.0182463
$R = 1$, $S3$	13	24	20	111.59	6.8	0.0289	0.00726931

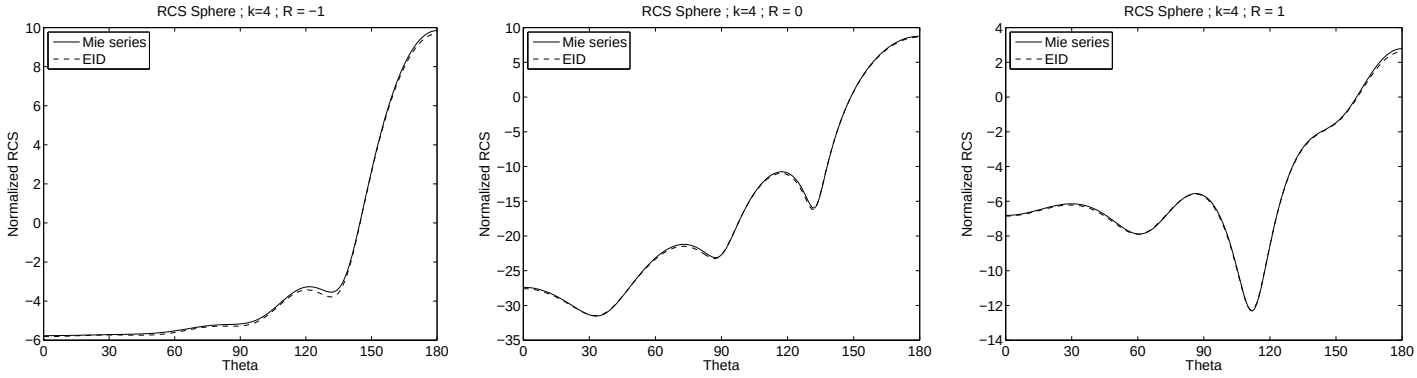

FIG. 2 – Cas où $k = 4$ et $R = -1\ ,\ 0\ ,\ 1$.

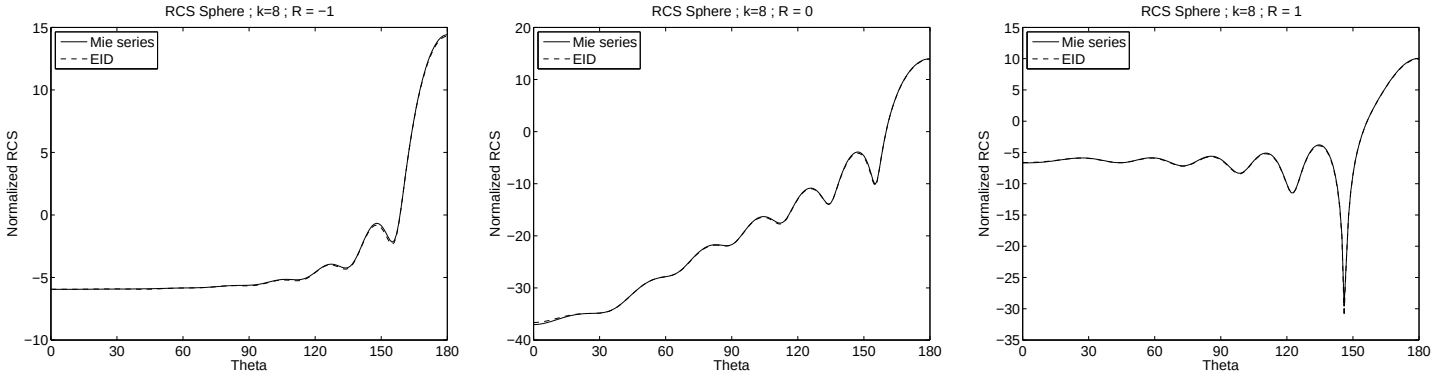

FIG. 3 – Cas où $k = 8$ et $R = -1\ ,\ 0\ ,\ 1$.

TAB. 2 – Coûts de résolution – cas $k = 4$, $k = 8$

Cas	nbCGi	nbCGo	nbJ	Tps (s)	Mem (Mo)	ErrRel	Y-iX
$R = -1$, $k = 2$	13	24	18	108	6.8	0.0229	0.0118533
$R = -1$, $k = 4$	16	28	12	1544	72	0.0158	0.0073715
$R = -1$, $k = 8$	18	24	9	22894	$1.1\ 10^3$	0.0119	0.0054462
$R = 0$, $k = 2$	14	27	–	87	6.8	0.0231	0.0056842
$R = 0$, $k = 4$	19	28	–	1256	72	0.0133	0.0043134
$R = 0$, $k = 8$	24	29	–	19643	$1.1\ 10^3$	0.0104	0.0034581
$R = 1$, $k = 2$	13	24	20	112	6.8	0.0289	0.00726931
$R = 1$, $k = 4$	16	27	24	1840	72	0.0128	0.00519959
$R = 1$, $k = 8$	17	26	29	31216	$1.1\ 10^3$	0.0090	0.00432662

5 TP2 – atelier MÉLINA 2009

En guise de deuxième TP de l'atelier MÉLINA 2009, nous vous proposons de compléter l'exemple `$eid/eid.c++` fourni avec la librairie MÉLINA++. Vous trouverez dans ce fichier un ensemble de trous à compléter selon les consignes données dans le fichier c++ aux endroits désignés par "`TP2 -- Atelier Melina 2009`".

Conclusion

Cet exemple met en œuvre et valide la plupart des opérateurs intégraux de surface fournis par la librairie MÉLINA++. Cependant, la librairie n'intègre pas, à ce jour, de méthodes de préconditionnement ou de méthodes rapides pour les équations intégrales mais cela fait évidemment partie des perspectives justifiant notamment certaines interventions de l'atelier MÉLINA 2009.

Références

[1] D. Martin. `http://anum-maths.univ-rennes1.fr/melina++`

[2] R. Kress. Linear Integral Equations, 2nd ed., Springer Verlag, 1999.

[3] D. Colton and R. Kress. Inverse Acoustic and Electromagnetic Scattering Theory, 2nd ed., Springer Verlag, 1998.

[4] J. C. Nédélec. Acoustic and Electromagnetic Equations, Integral Representations for Harmonic Problems, Springer-Verlag, 2000.

[5] B. Després. Fonctionnelle quadratique et équations intégrales pour les problèmes d'onde harmonique en domaine extérieur, M2AN, vol. 31 (6), 679–732, 1997.

[6] J.-D. Benamou and B. Després. A domain decomposition method for the Helmholtz equation and related optimal control problems, J. Comput. Phys., vol. 136 (1), 68–82, 1997.

[7] N. Bartoli and F. Collino. Integral Equations via Saddle Point Problem for Acoustic Problems, M2AN, vol. 34 (5), 1023–1049, 2000.

[8] B. Stupfel, A Hybrid Finite Element and Integral Equation Domain Decomposition Method for the Solution of the 3-D Scattering Problem, J. Comput. Phys., vol. 172, pp. 451–471, 2001.

[9] E. Darrigrand. Coupling of Fast Multipole Method and Microlocal Discretization for the 3-D Helmholtz Equation, J. Comput. Phys., vol. 181 (1), pp. 126–154, sept. 2002.