Résolution d'équations aux dérivées partielles non linéaires et couplées

Stéphanie Marchesseau Dimitri Bettebghor
Directeur de projet : Pierre Dreyfuss

Ecole des Mines de Nancy, 2006-2007

Table des matières

Avertissement/Remerciements

Le contenu de ce rapport appartient à ses auteurs, l'Ecole des Mines de Nancy n'est en rien responsable des opinions exprimées.

Nous remercions chaleureusement notre directeur de projet M.Pierre Dreyfuss, chercheur à l'Institut Elie Cartan de l'Université Henri Poincaré de Nancy. Ses conseils et son aide ont été très précieux tout au long de ce projet.

Introduction

Notre projet que l'on pourrait définir comme un projet de calcul scientifique, a ses applications dans le domaine physique de l'électromagnétisme, puisqu'il consiste à déterminer les grandeurs représentatives d'un aimant : l'aimant de Bitter. Nous présenterons donc brièvement les caractéristiques d'un tel aimant, avant de nous intéresser plus particulièrement aux équations qui régissent son fonctionnement, équations que nous essaierons de résoudre par la simulation numérique.

En effet, le point le plus important de notre projet fut de traiter des équations différentielles non linéaires et couplées. Cela signifie qu'il nous a été impossible de nous aider des méthodes habituelles de résolution des équations différentielles, c'est pourquoi nous avons dû envisager de transformer notre problème afin de pouvoir le résoudre par des méthodes bien connues, comme la méthode de Newton.

La méthode de Newton permet de trouver le zéro d'une fonction, il a donc fallu passer de deux équations différentielles non linéaires couplées à un problème de point fixe, nous vous présenterons bien sûr la théorie qui se cache derrière cette transformation. Ensuite, nous nous sommes attaqués à la manière de résoudre numériquement ce nouveau problème, par la programmation (avec Matlab) de multiples fonctions imbriquées, dont nous expliquerons l'intérêt par la suite.

Le programme étant plutôt compliqué, et voulant nous assurer de sa pertinence, nous avons envisagé un cas test, dont nous connaissions le résultat théorique. Après de nombreuses heures passées sur ce sujet, nos efforts ont finalement été récompensés puisque nous sommes aujourd'hui en mesure de fournir un programme satisfaisant, qui répond à la problématique que nous avions fixée au début de notre projet. On pourrait évidemment continuer à creuser le sujet, comme par exemple se ramener à de exemples physiques et interpréter les résultats, mais le temps nous a manqué, nous laissons donc cela à de potentiels successeurs.

Définitions et notations

$\mathbb{N}$ Ensemble des entiers naturels

$\mathbb{R}$ Ensemble des réels

∇ Opérateur gradient. Soit u une fonction de $\mathbb{R}^n$ dans $\mathbb{R}$, le gradient, noté aussi grad $u(x)$ est :

$$\nabla u(x) = (\frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n})$$

div Opérateur divergence. Soit $V = (V_1, \ldots, V_n)$ une fonction de $\mathbb{R}^m$ dans $\mathbb{R}^n$, div V est :

$$\text{div}V(x) = \frac{\partial V_1}{\partial x_1} + \ldots + \frac{\partial V_n}{\partial x_n}$$

Δ Opérateur laplacien. Soit u une fonction, le laplacien est :

$$\Delta u(x) = \frac{\partial^2 V_1}{\partial x_1^2} + \ldots + \frac{\partial^2 u}{\partial x_n^2} = \text{div}(\nabla u(x))$$

$H^1(\Omega)$ Espace de Sobolev. Soit Ω un ouvert de $\mathbb{R}^n$, on définit $H^1(\Omega)$ comme :

$$H^1(\Omega) = (u \in L^2(\Omega) / \frac{\partial u}{\partial x_i} \in L^2(\Omega) \text{pour } i = 1 \ldots n)$$

$H_0^1(\Omega)$ Noyau de l'opérateur linéaire trace, noté τ_0 de $H^1(\Omega)$ dans $L^2(\partial\Omega)$

$H^{1/2}(\partial\Omega)$ Image de l'espace $H^1(\Omega)$ par l'application trace

Chapitre 1

Présentation du problème

On commence par introduire les équations aux dérivées partielles de manière générale, puis on présente le sujet de notre projet et le cadre physique dont il est issu.

1.1 Les équations aux dérivées partielles

1.1.1 Equations aux dérivées partielles linéaires

L'un des enjeux les plus importants du calcul scientifique est la résolution d'équations aux dérivées partielles (qu'on abrègera en E.D.P tout au long de ce rapport). En effet, de très nombreux problèmes issus des sciences physiques (physique quantique, électromagnétisme, conductivité thermique) de la mécanique (mécanique des fluides, mécanique du solide, élasticité) de la chimie (mécanismes de réaction-diffusion) ou encore de la biologie (dynamique des populations) et de la finance (pricing d'actifs) sont décrits à l'aide des E.D.P. Une E.D.P fournit une relation entre les dérivées partielles d'une fonction de plusieurs variables (qui peut ou non dépendre du temps). Parmi les E.D.P les plus classiques on trouve les suivantes :

L'équation de la chaleur que l'on rencontre en conductivité thermique. Cette équation décrit l'évolution de la température T par rapport à une source de chaleur notée f :

$$\frac{\partial T}{\partial t} - \Delta T = f$$

L'équation des ondes que l'on rencontre en élasticité et en électromagnétisme, qui relie le déplacement d'une membrane (ou d'une onde) u sous l'effet d'une force f :

$$\frac{\partial^2 u}{\partial^2 t} - \Delta u = f$$

L'équation des télégraphes que l'on rencontre en électromagnétisme. Cette d'équation décrit l'évolution du potentiel V en fonction du temps et de sa position au sein d'un matériau (ici dans le cas mono-dimensionnel) :

$$\frac{\partial^2 V}{\partial^2 t} - \alpha \frac{\partial V}{\partial t} - \frac{\partial^2 V}{\partial^2 x} = 0$$

Bien sûr, un problème d'E.D.P n'est clairement posé que lorsqu'on s'est donné un ouvert Ω de $\mathbb{R}^2$ ou de $\mathbb{R}^3$ sur lequel on a l'E.D.P proprement dite et la frontière $\partial\Omega$ de cet ouvert sur laquelle on se donne des conditions au bord et, si le temps intervient, des conditions initiales. On ne sait pas donner les solutions exactes des équations ci-dessus dans bien des cas, c'est pourquoi on a recours à des méthodes d'approximation numérique (méthodes des éléments finis ou des différences finies). Néanmoins, les équations ci-dessus sont relativement simples dans la mesure où il s'agit d'E.D.P linéaires : c'est-à-dire que les relations entre les dérivées partielles sont linéaires (soit on les additionne ou on les multiplie par un scalaire ou bien par une fonction indépendante de la solution cherchée).

1.1.2 Equations aux dérivées partielles non linéaires

Dans bien des cas, on rencontre des E.D.P non linéaires, c'est-à-dire que la relation entre les dérivées partielles est non linéaire. Par exemple elle fait intervenir le carré d'une dérivée ou bien on multiplie par une fonction qui dépend elle-même de la solution. Voici quelques unes d'entre elles :

L'équation iconale que l'on rencontre en optique qui décrit l'évolution d'un champ monochromatique S :

$$||\nabla S|| = 1$$

L'équation de Burgers non visqueuse que l'on rencontre en mécanique des fluides qui décrit l'évolution du déplacement u au sein d'un fluide (cas mono-dimensionnel) :

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

Et bien sûr, les célèbres **équations de Navier-Stokes** de la mécanique des fluides. Equations qui rappelons-le, sont loin d'avoir livré tous leurs secrets. Le *Clay Mathematics Institut* offre 1 000 000 de dollars à quiconque fait avancer leur difficile théorie.

La plupart du temps, on ne sait pas résoudre de manière exacte ces E.D.P non linéaires. Pourtant leur résolution est cruciale tant du point de vue théorique qu'en vue de leurs applications industrielles. Notre projet consistait à étudier et résoudre des E.D.P non linéaires.

1.2 Présentation de notre problème d'E.D.P

Le problème traité dans ce projet s'inscrit dans la résolution numérique d'équations aux dérivées partielles. On s'est attelé à résoudre un système de deux équations aux dérivées partielles elliptiques non linéaires couplées entre elles, équations issues à la fois de la théorie de l'électromagnétisme et de la théorie de la conduction thermique. Ces équations se rencontrent dans les électroaimants de Bitter que l'on présente brièvement.

1.2.1 Les électroaimants de Bitter

Les électroaimants de Bitter permettent de produire des champs magnétiques intenses (les plus récents atteignent 60 Tesla). L'innovation principale de ces électroaimants (due à Francis Bitter, physicien américain) est d'utiliser non plus des spires pour produire des champs mais des couronnes ou des disques de cuivre disposés de manière hélicoïdale (ces disques sont d'ailleurs appelés *Bitter plates*) séparés par des isolants. Comme on peut le voir sur les schémas (Fig. 1.1 et 1.2),

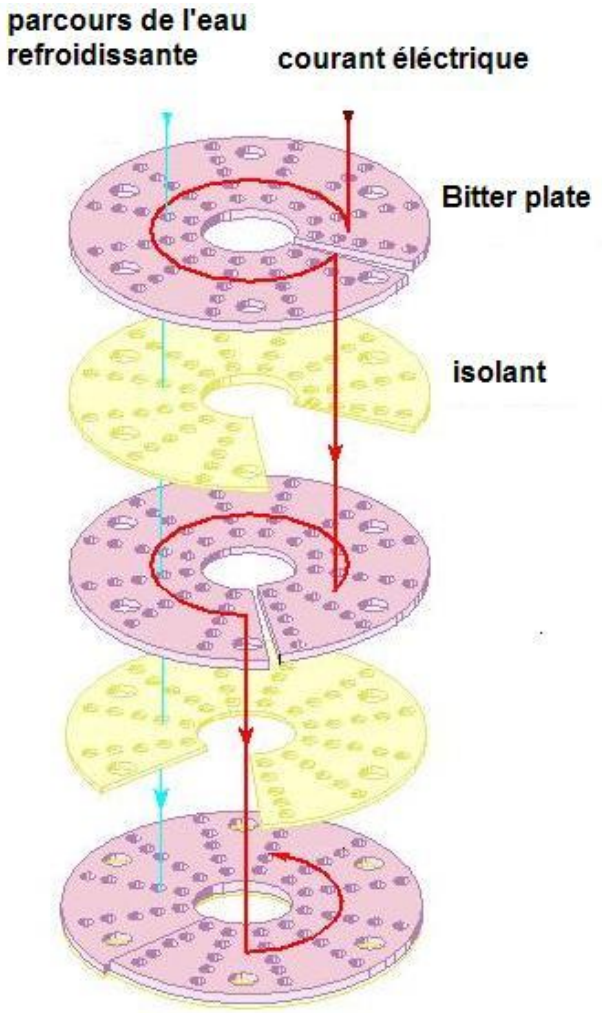


FIG. 1.1 – Schéma de fonctionnement de l'électroaimant

ces *Bitter plates* sont trouées. Ces trous permettent de faire circuler de l'eau (en générale déionisée et à environ $10^{\circ}C$) de manière à refroidir la structure qui atteint des températures très élevées à de tels champs magnétiques. En effet, la chaleur produite augmente comme le carré du champ magnétique. Le passage de l'eau permet de maintenir une température de l'ordre de $30^{\circ}C$ dans une grande partie de l'aimant. De plus, la structure doit être particulièrement robuste pour résister aux forces de pression dûes à la force de Lorentz qui elle aussi augmente avec le carré du champ.

De tels électroaimants sont utilisés à des champs que ne permettent pas de créer les supraconducteurs. En effet, pour la plupart des matériaux supraconducteurs, la supraconductivité cesse au-delà de champs magnétiques de l'ordre de 10 Tesla.

1.2.2 Problème théorique

Notre but est de déterminer le potentiel V et la température T au sein d'une *bitter plate* soumis à un potentiel V_0. On se place dans l'hypothèse de régime permanent, c'est-à-dire que ces deux quantités ne varient pas avec le temps et on cherche à établir les équations qui les lient.

On se place donc sur un ouvert Ω de $\mathbb{R}^2$ représentant la *Bitter plate*, cette *Bitter plate* est caractérisée par deux grandeurs couplées dépendants de la température : la conductivité électrique et

la conductivité thermique

FIG. 1.2 – Bitter plate

la conductivité électrique σ : La conductivité électrique est l'aptitude d'un matériau à laisser les charges électriques se déplacer librement,c'est-à-dire permettre le passage du courant électrique. Elle est exprimée en Siemens par Mètre . Elle est définie de la manière suivante :

$$\begin{array}{ll} \sigma(T) = \sigma_0 & \forall T \leq T_0 \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \geq T_c \end{array}$$

avec T_0 une température de référence, T_c une température critique au-delà de laquelle le système n'a plus de sens physique, et σ_0 et $\alpha \in \mathbb{R}$ qui contrôle la dépendance de σ par rapport à T.

la conductivité thermique κ : la conductivité thermique est une mesure physique caractérisant le comportement des matériaux lors du transfert de chaleur par conduction. Elle représente la quantité de chaleur transférée par unité de surface et par unité de temps sous un gradient de température. Dans le système international d'unités, la conductivité thermique est exprimée en Watts par Mètre-Kelvin. Ces deux conductivités sont reliées pour chaque métal par la loi de Wiedemann-Frantz :

$$\frac{\kappa}{\sigma T} = L$$

où L est le nombre de Lorentz (de l'ordre de 10^{-8}). D'où :

$$\begin{array}{ll} \kappa(T) = L\sigma_0 T_0 & \forall T \leq T_0 \\ \kappa(T) = L\sigma(T)T = \frac{L\sigma_0 T}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \kappa(T) = L\sigma_c T_c & \forall T \geq T_c \end{array}$$

La densité de courant $\vec{j}$ au sein de Ω est alors :

$$\forall x \in \Omega, \quad \vec{j}(x) = \sigma(T)\nabla(V(x)) \tag{1.1}$$

On obtient la première E.D.P sur le potentiel V en appliquant l'équation locale de conservation de la charge issue des équations de Maxwell :

$$\mathrm{div}(\vec{j}) + \frac{\partial \rho}{\partial t} = 0$$

où ρ désigne la densité de charge électrique au sein du matériau. Comme on se place en régime permanent, l'équation devient : $\text{div}(\vec{j}) = 0$ soit, dans le cas des aimants de Bitter :

$$\text{div}(\sigma(T)\nabla(V)) = 0 \tag{1.2}$$

Pour obtenir l'équation sur la température T, on utilise la loi de Fourier :

$$\text{div}(\kappa(T)\nabla T) + P = \rho c \frac{\partial T}{\partial t}$$

où κ désigne la conductivité thermique, ρ la masse volumique du matériau, c la chaleur massique du matériau et P est l'énergie produite au sein du matériau, elle peut avoir différentes origines. Comme on se place en régime permanent, l'équation devient :

$$\text{div}(\kappa(T)\nabla T) + P = 0$$

Reste à déterminer P l'énergie produite au sein du matériau par le champ magnétique : $P = RI^2$ où R désigne la résistivité du matériau, inverse de la conductivité et I l'intensité, on sait que : $i = \vec{j}dS$ où i est le courant électrique et dS un morceau élémentaire de surface, dans notre cas, on a donc :

$$P = \frac{1}{\sigma(T)}(\sigma(T)\nabla(V))^2$$

soit finalement :

$$P = \sigma(T)||\nabla(V)||^2 \tag{1.3}$$

On a donc établi nos deux E.D.P couplées au sein du domaine Ω :

$$\begin{cases} \text{On cherche (V,T) tels que :} & \\ -\text{div}(\sigma(T)\nabla(V)) = 0 & \text{dans } \Omega \\ -\text{div}(\kappa(T)\nabla(T)) = \sigma(T)||\nabla(V)||^2 & \text{dans } \Omega \end{cases}$$

Restent alors les conditions au bord, puisqu'on se place en régime permanent il n'y a pas de conditions initiales. Il faut maintenant distinguer deux types de conditions au bord. En effet, la frontière se divise en deux sous-domaine distincts : $\partial\Omega = \Gamma = \Gamma_D \cup \Gamma_N$ où Γ_D désigne le bord des rainures qui coupent en deux la largeur du disque et Γ_N désigne le bord des trous internes et les bords externes de la couronne.

Le potentiel V est fixé sur le bord des rainures à la valeur V_0, on a donc la condition :

$$V = V_0 \quad \text{sur } \Gamma_D \tag{1.4}$$

De plus, on considère que le flux de densité de courant $\vec{j}$ est nul sur Γ_N :

$$-\sigma(T)\nabla(V).\vec{n} = 0 \quad \text{sur } \Gamma_N \text{ où } \vec{n} \text{ désigne le vecteur normal à la frontière} \tag{1.5}$$

Enfin, on a une condition mixte qui mêle la température et sa dérivée normale (produit scalaire du gradient et du vecteur normal) sur toute la frontière Γ, en notant T_W la température de l'eau et $\beta \in \mathbb{R}$ on a :

$$-\kappa(T)\nabla T.\vec{n} + \beta T = \beta T_W \quad \text{sur } \Gamma \tag{1.6}$$

Pour résumer, notre problème est alors le suivant :

$$\begin{cases} \text{On cherche (V,T) tels que :} & \\ -\text{div}(\sigma(T)\nabla(V)) = 0 & \text{dans } \Omega \\ -\text{div}(\kappa(T)\nabla(T)) = \sigma(T)||\nabla(V)||^2 & \text{dans } \Omega \\ V = V_0 & \text{sur } \Gamma_D \\ -\sigma(T)\nabla(V).\vec{n} = 0 & \text{sur } \Gamma_N \\ -\kappa(T)\nabla T.\vec{n} + \beta T = \beta T_W & \text{sur } \Gamma \end{cases}$$

La première condition au bord est appelée "condition de Dirichlet inhomogène", la seconde "condition de Neumann pure" et la dernière "condition mixte" ou "condition de Fourier-Robin". Il s'agit là des trois types de conditions au bord les plus classiques.

Chapitre 2

Quelques précisions sur Matlab

Tout au long de ce projet, nous nous sommes aidés du logiciel de calcul Matlab (abrégé de *Matrix Laboratory*) qui permet de mener à bien des calculs principalement matriciels mais qui fournit aussi des algorithmes efficaces d'approximations numériques (intégrales, racines de polynôme...). Le but de cette partie est d'éclairer et préciser les fonctionnalités de Matlab et plus précisément celles relevant de la géométrie, des éléments finis et des équations aux dérivées partielles, mais aussi celles relatives au stockage des données. Ceci en vue de permettre une meilleure compréhension des algorithmes que nous développons par la suite.

2.1 La PDE Toolbox de Matlab et stockage de données

La `PDE Toolbox` de Matlab est un module de Matlab qui offre à la fois tout un panel de commandes principalement dédiées à la résolution, à l'aide de la méthode des éléments finis, des équations aux dérivées partielles linéaires en deux dimensions et dans une moindre mesure celles non linéaires, mais elle fournit de plus un environnement graphique pratique et des outils typiquement géométriques intervenant dans la résolution des équations aux dérivées partielles par la méthode des éléments finis (maillage, rafinement, décomposition de domaine...). Ces fonctionnalités utilisent un codage de la géométrie qu'il nous a fallu décrypter avant d'écrire nos premiers algorithmes.

2.1.1 L'interface graphique (*Graphic User's Interface*)

La G.U.I permet de créer le domaine sur lequel on cherche à résoudre l'E.D.P à l'aide de commandes classiques de logiciels de dessin (rectangles, lignes brisées, ellipses). On peut donc dessiner un large éventail de domaines différents, bien qu'on soit limité aux domaines dans $\mathbb{R}^2$. Signalons au passage que des logiciels professionnels type *Comsol Multiphysics* ou *Cesar* offrent bien plus de possibiltés quant à la création de domaines et les équations traitées. De plus, on ne peut pas tracer de courbes pour des domaines "exotiques" (frontières paraboliques ou hyperboliques par exemple). Néanmoins, à l'aide de la combinaison de plusieurs domaines on a pu tracer le domaine simplifié correspondant à notre problème (couronne "trouée", voir Fig 2.1 page suivante).

Enfin, il faut préciser que toutes les opérations que l'on effectue dans l'interface graphique sont accessibles en ligne sous Matlab, c'est pourquoi on précise la plupart du temps les commandes, ce qui peut être utile quand on veut résoudre une E.D.P sur un domaine que l'on ne peut pas dessiner sous cette interface. Une fois le domaine tracé, on commence le maillage du domaine. Ici encore, la PDE Toolbox se révèle parfois insuffisante car les seuls maillages que l'on peut réaliser à partir de l'interface graphique sont des maillages de Delaunay triangulaire, alors que dans certains cas, on

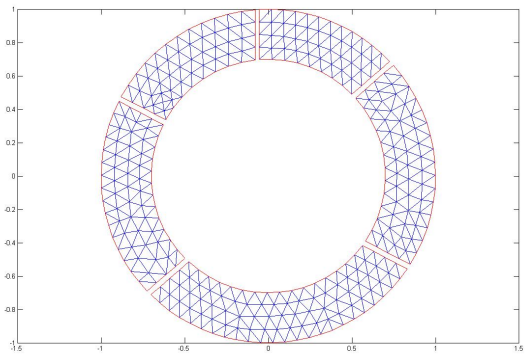


FIG. 2.1 – Domaine simplifié Ω créé dans l'interface graphique

préfère des maillages différents (plus réguliers comme le maillage de Poisson par exemple ou bien des maillages quadrangles selon la géométrie du domaine). De plus, on ne peut pas non plus obtenir des maillages adaptatifs ou anisotropiques où le nombre d'éléments est bien plus élévés au voisinage de singularités ou bien en des endroits où l'on a besoin de plus de précisions sur la solution, comme par exemple dans le cas d'une coque de bateau ou d'une aile d'avion où les maillages ressemblent à celui-ci :

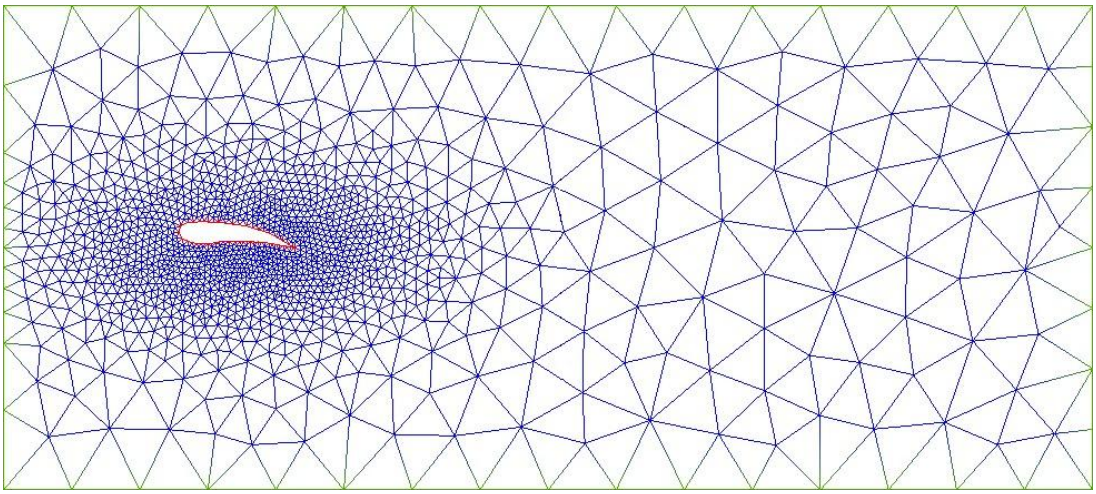


FIG. 2.2 – Maillage adaptatif d'une aile

En définitive, le mailleur de la PDE Toolbox ne permet pas une grande liberté quant au choix du maillage. Ce qui dans certains cas peut s'avérer dommageable sur la précision de la solution. Une fois le maillage initialisé (ce qu'on peut obtenir en ligne à l'aide de la commande `initmesh`), on peut décider de le raffiner c'est-à-dire d'augmenter le nombre d'éléments ou ce qui revient au même de diminuer le pas de discrétisation pour obtenir une meilleure approximation de la solution. En ligne, on utilise la commande `refinemesh`. Enfin, on peut améliorer la qualité du maillage à l'aide de la commande `jigglemesh` qui a pour fonction de "régulariser" le maillage. En effet, des triangles trop aplatis ou de tailles disparates nuisent à la résolution en augmentant le conditionnement de la matrice de rigidité du problème associé (*stiffness matrix* sous Matlab et généralement notée K). Voici un exemple de maillage obtenu grâce à la PDE Toolbox où on a affiché les numéros des noeuds et des triangles (Fig. 1.3 page suivante).

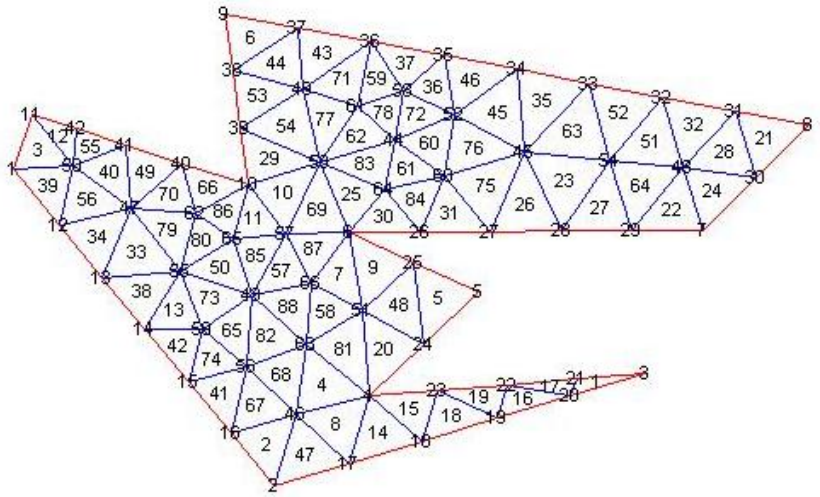


FIG. 2.3 – Exemple de maillage et de numérotation

De même que le choix du maillage est extrêmement important pour la résolution des E.D.P, la numérotation des noeuds et des triangles est aussi cruciale car le caractère "régulier" de la matrice de rigidité en dépend essentiellement. Une numérotation judicieuse conduit à une matrice plus régulière pour laquelle on peut développer des algorithmes spécifiques permettant une résolution du système linéaire plus efficace (matrice tridiagonale, matrice bande...) au contraire, une mauvaise numérotation peut conduire à une matrice sans spécificité particulière et donc entraver la résolution, bien que le nombres d'élements de la matrice soit le même. Il existe des algorithmes pour améliorer la numérotation qui ne sont pas développés dans la PDE Toolbox, de plus, la numérotation est fixée dans l'interface graphique, on ne peut pas la modifier, sauf si on entre soi-même en ligne la géométrie du domaine et le maillage.

De plus il faut entrer les conditions au bord du domaine (Dirichlet, Neumann, Fourier-Robin). On entre alors dans le `boundary mode` qui affiche la frontière du domaine crée. Malheureusement, la PDE Toolbox choisit le découpage de la frontière qui lui convient le mieux. Par exemple, la frontière d'un cercle ou d'une ellipse est divisée en quatre cadrants et on spécifie la condition au bord sur chacun, ou bien le découpage de la frontière d'un polygone est tout simplement le découpage selon ses côtés. Si l'on a des conditions au bords plus "exotiques" (Dirichlet sur les deux tiers de la circonférence et Neumann sur le tiers restant par exemple) on ne peut pas les rentrer sous l'interface graphique. Encore une fois, on est obligé de les rentrer en ligne "à la main".

Enfin, la dernière lacune de la PDE Toolbox réside dans la méthode des éléments finis elle-même. En effet, tous les algorithmes de Matlab se fondent sur une méthode des éléments finis $\mathbb{P}1$, c'est-à-dire que les fonctions de base ϕ_i sont affines, alors que l'on a parfois besoin de fonctions polynomiales d'ordre 2 ou parfois d'ordre supérieur, ceci pour obtenir une meilleur approximation ou encore une meilleure convergence du gradient de la solution si c'est ce dernier qui nous intéresse.

On l'a vu, l'interface graphique de la PDE Toolbox bien que très simple à prendre en main souffre

de nombreuses lacunes pour qui désire résoudre une E.D.P sur un domaine particulier, en choisissant le maillage ou encore avec des conditions au bord inhabituelles. On peut surmonter quelques unes de ces lacunes en entrant directement les objets informatiques que Matlab utilise en ligne. Pour ce faire, il faut comprendre comment Matlab code la géométrie, le maillage et les conditions au bord pour pouvoir profiter pleinement des fonctionnalités de la PDE Toolbox. C'est l'objet de la partie suivante.

2.1.2 Codage de la géométrie et du maillage

Une fois la géométrie du domaine, le maillage, les conditions au bords et l'équation aux dérivées partielles à résoudre entrés sous l'interface, on peut résoudre directement l'équation sous l'interface ou bien exporter toutes les données de l'équation sous Matlab dans le `Workspace` pour travailler sur ces données et résoudre de manière plus approfondie l'équation. C'est ce qui nous a permis de comprendre comment Matlab code les informations à propos de l'E.D.P à résoudre. Les caractéristiques de l'équation sont codées sous forme de matrices parfois obscures, en voici quelques unes décodées sur l'exemple suivant où l'on cherche à résoudre l'équation de Laplace avec condition de Dirichlet homogène sur la frontière du domaine suivant :

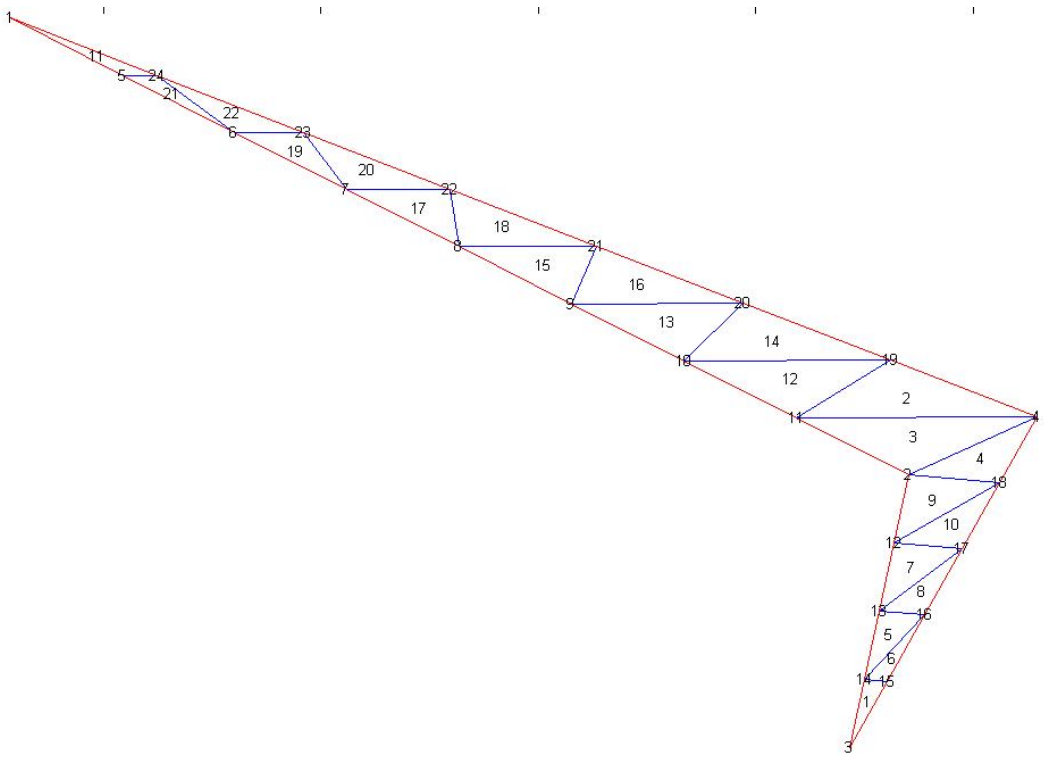


FIG. 2.4 – Exemple de domaine

Conditions au bord et géométrie décomposée

Ces conditions sont codées au moyen de deux matrices, généralement notés `b` et `g`. Pour notre exemple, on a pris une condition de Dirichlet homogène sur le bord. Par contre, comme précisé plus haut, on ne peut pas intervenir dans la PDE Toolbox pour modifier la décomposition géométrique du domaine. C'est-à-dire que si on utilise telles quelles les données exportées, on ne peut pas intervenir dessus.

La matrice `g` : La matrice `g` code les caractéristiques géométriques de la frontières du domaine. Matlab réalise en fait la décomposée géométrique du domaine, c'est-à-dire que la frontière est décomposée en "morceaux" élémentaires sur lesquelles on spécifie une condition au bord. Ces "morceaux" peuvent être des arcs d'ellipses, des segments. Mais cette opérations est la première à être effectuée car comme on le verra plus bas, la numérotation des noeuds du maillage dépend de ces briques élémentaires. En effet, Matlab cherche à rendre la numérotation efficace et cette décomposition favorise une numérotation judicieuse. Dans notre exemple, Matlab choisit automatiquement de diviser la frontière du domaine en quatre briques, ici il s'agit des arêtes du polygones. A chaque colonne de la matrice `g` correspond une de ces briques. Pour le domaine considéré ici, la matrice aura donc 4 colonnes. Les lignes sont au nombre de 7 et se remplissent comme suit :

1. le "type" géométrique du sous domaine de la frontière (arcs de cercle ou d'ellipse, segment) : 1 pour un arc de cercle, 2 pour un segment, 4 pour un arc d'ellipse.
2. les abscisses du point de départ et du point d'arrivée du sous domaine (deux lignes)
3. les ordonnées du point de départ et du point d'arrivée du sous domaine (deux lignes)
4. les labels des régions, c'est-à-dire le numéro de la région à gauche puis de celle à droite, avec la convention 1 si l'on est à l'intérieur du domaine et 0 si l'on est à l'extérieur.

On obtient alors la matrice suivante dans notre exemple :

$$\mathtt{g} = \begin{pmatrix} 2 & 2 & 2 & 2 \\ -0.27154 & -0.064877 & -0.078297 & -0.035354 \\ -0.064877 & -0.078297 & -0.035354 & -0.27154 \\ 0.50028 & 0.13901 & -0.075975 & 0.1848 \\ 0.13901 & -0.075975 & 0.1848 & 0.50028 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

La matrice `b` : La matrice `b` code les conditions au bord. Elle a été la plus longue à comprendre car elle fait intervenir du code de caractère `ASCII`. Une fois la décomposée géométrique effectuée, on a nos "morceaux" de frontière sur lesquels on va définir les conditions au bord. Chaque colonne de la matrice `b` représente la condition sur un sous domaine de la frontière. Elle a donc autant de colonnes que la matrice `g`. Pour ce qui est de la définition des conditions proprement dites, Matlab part des équations suivantes (où u est la solution du problème) :

$$\begin{cases} \vec{n}.\nabla u + qu = g & \text{où } \vec{n} \text{ est le vecteur normal à la frontière et } q \text{ et } g \text{ sont des fonctions} \\ hu = r & \text{où } h \text{ et } r \text{ sont des fonctions} \end{cases}$$

De cette manière, on peut décrire la plupart des conditions au bord, Dirichlet grâce à la deuxième équation, Neumann pure grâce à la première en annulant q et Fourier-Robin grâce à la première, juste en annulant certains coefficients de l'équations et donnant les valeurs des autres. Ainsi, les seules véritables données pour chaque sous domaine de la frontière sont ces q,g,h et r. Comme il s'agit en général de fonctions (mais on peut les rentrer aussi sous forme de vecteurs), Matlab les traduit en chaînes de caractères (format `string`) ensuite traduites en `ASCII`. La matrice `b` comporte 10 lignes remplies ainsi :

1. dimension du sous domaine considéré (ici 1 puisqu'il s'agit de segments)

2. nombre de conditions aux bord sur le sous domaine considéré. On ne peut en donner qu'une si l'on passe par l'interface graphique.
3. les quatre lignes suivantes comportent la longueurs des chaînes de caractères ASCII décrivant les fonctions q,g,h et r.Ici, on a pris des conditions particulièrement simples, ces fonctions sont toutes nulles à l'exception de h qui vaut 1. Les chaînes sont donc celles traduisant 1 ou 0, elles sont toutes les deux de longueur 1.
4. les quatre dernières lignes sont les chaînes de caractères décrivant les fonctions q,g,h et r traduit en code `ASCII` : 48 (resp. 49) code le chiffre 0 (resp. 1).

Pour notre domaine, on obtient la matrice suivante :

$$\mathtt{b} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 48 & 48 & 48 & 48 \\ 48 & 48 & 48 & 48 \\ 49 & 49 & 49 & 49 \\ 48 & 48 & 48 & 48 \end{pmatrix}$$

Le maillage

Quand on crée un maillage et qu'on l'exporte sous Matlab, on obtient trois matrices généralement notées `p`,`e` et `t`. Ces trois matrices renferment toutes les informations relatives au maillage. Le maillage de ce domaine comprend 24 noeuds numérotés de 1 à 24 et 22 triangles numérotés de 1 à 22.

La matrice `p` : La matrice `p` (`p` pour *points*) a 2 lignes et 24 colonnes (soit le nombre de noeuds). Il s'agit des coordonnées de chacun des points du maillage rangés selon leur numérotation. La ligne 1 est celle des abscisses et la ligne 2 celle des ordonnées. Pour ce qui est de la numérotation, on peut noter que Matlab numérote en premier les noeuds qui se trouvent aux frontières de la décomposition géométrique (ici ce sont les sommets du polygone), puis numérote de manière croissante en parcourant tout le reste de la frontière :

$$\mathtt{p} = \begin{pmatrix} -0.27154 & -0.064877 & \dots & -0.2378 \\ 0.50028 & 0.13901 & \dots & 0.45521 \end{pmatrix}$$

La matrice `t` : La matrice `t` (`t` pour *triangles*) a 4 lignes et 22 colonnes (soit le nombre de triangles). Les trois premières lignes comprennent les numéros des trois sommets de chaque triangle, les triangles étant rangés selon leur numérotation. La dernière ligne est un booléen qui indique si le triangle se trouve à la frontière du domaine (ici c'est la cas de tous les triangles). Par contre, l'ordre de numérotation n'apparaît pas clairement, de même que l'ordonnancement des trois sommets ne paraît pas être régi selon une règle spéciale :

$$\mathtt{t} = \begin{pmatrix} 14 & 11 & \dots & 6 \\ 3 & 4 & \dots & 23 \\ 15 & 19 & \dots & 24 \\ 1 & 1 & \dots & 1 \end{pmatrix}$$

La matrice `e` : La matrice `e` (`e` pour *edges*) ne prend en compte que les arêtes qui forment la frontière du domaine, les arêtes internes ne sont représentées dans cette matrice. La matrice `e` a 7 lignes et 24 colonnes (soit le nombre d'arêtes qui forment la frontière). Les arêtes sont placées à la suite les unes des autres comme sur le domaine, elles commencent au noeud numéro 1 et parcourent la frontière jusqu'à revenir au noeud numéro 1 (dans notre exemple la première arête est celle reliant 1 à 5, puis 5 à 6, 6 à 7,...jusqu'à l'arête reliant 24 à 1) de manière à avoir à la suite les arêtes du même sous domaine de la frontière. Les 7 lignes sont remplies de la manière suivante :

1. le numéro du noeud où commence l'arête
2. le numéro du noeud où s'achève l'arête
3. la longueur du sous domaine parcouru avant d'arriver à l'arête ramené à 1 où 1 est la longueur cumulée des arêtes sur tout un sous domaine.
4. la longueur du sous domaine quand l'arête se termine.
5. le label (i.e le numéro) du sous domaine considéré (ici on en a 4).
6. les deux dernières lignes sont formées de booléens qui indique la place du domaine par rapport à l'arête.

Pour le domaine considéré on a donc la matrice suivante :

$$\mathtt{e} = \begin{pmatrix} 1 & 5 & \dots & 23 & 24 \\ 5 & 6 & \dots & 24 & 1 \\ 0 & 0.125 & \dots & 0.71429 & 0.85714 \\ 0.125 & 0.25 & \dots & 0.85714 & 1 \\ 1 & 1 & \dots & 4 & 4 \\ 1 & 1 & \dots & 1 & 1 \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

2.1.3 Stockage des données : le format `sparse`

Comme on le voit dans l'exemple précédent, les objets (principalement des matrices) qui codent toutes les caractéristiques d'une E.D.P et qui en permettent le traitement par la méthode des éléments finis sont de grande taille. Par exemple, si on raffine une seule fois le carré unité, le nombre de noeuds du maillage est de 665, ce qui signifie que les matrices de rigidité K et de masse M associées sont de taille $665^2 = 442225$ ce qui rend la résolution des systèmes passablement longue, de plus, ces matrices occupent un grand espace mémoire. Une manière de réduire l'espace occupé par ces matrices et de faciliter leur traitement informatique est proposé par le format `sparse` ("creux") présent dans Matlab.

Le format `sparse` permet de prendre en compte le grand nombre de coefficients nuls présents dans une matrice. En effet, les matrices utilisées dans la résolution d'E.D.P à l'aide des éléments finis ont en général peu d'éléments non nuls car un noeud n_i du maillage est connecté tout au plus à 5, 6 ou 7 éléments. Ainsi chaque ligne des matrices associées sont majoritairement vide. L'idée est donc de ne considérer que les coefficents non nuls puisque seuls ceux-ci interviennent dans les produits matrices-vecteurs ou matrices-matrices. On ne conserve que les coordonnées des éléments non nuls et leur valeur pour chaque couple de coordonnées les autres coefficients étant implicitement mis à 0. Plus précisément Matlab stocke les matrices `sparse` de la manière suivante :

1. avant de stocker les éléments non nuls, il faut décider d'un sens de parcours de ces éléments, plusieurs sens sont bien évidemment possibles : dans les sens des colonnes ou des lignes. Matlab les parcourt dans le sens des colonnes.
2. on stocke les abscisses des éléments non nuls dans un premier vecteur noté généralement `i` sous Matlab
3. on stocke les ordonnées des éléments non nuls dans un second vecteur noté `j`
4. enfin on stocke les valeurs des éléments non nuls dans le même ordre que leurs coordonnées dans un vecteur noté `s`

On peut créer une matrice `sparse` en définissant d'abord les vecteurs `i,j` et `s` puis en les assemblant à l'aide de la commande `sparse` (syntaxe `M=sparse(i,j,k)`) mais on peut aussi convertir une matrice en format `sparse` toujours à l'aide de la même commande (syntaxe `M=sparse(A)` où `A` est une matrice définie de manière classique). Enfin, on peut travailler directement sous forme `sparse` à l'aide de nombreuses commandes qui créent directement les matrices sous forme `sparse` : `spalloc` crée une matrice `sparse` en vue d'un remplissage futur, `speye` renvoie la matrice identité sous forme creuse, `spones` renvoie une matrice `sparse` où les éléments non nuls valent tous 1, `sprand` renvoie une matrice `sparse` où les coefficients non nuls sont distribués selon une loi uniforme sur $[0, 1]$ et placés de manière aléatoire au sein de la matrice, `spdiags` permet de générer des matrices `sparse` à partir de leurs diagonales. Enfin une commande très utile est fournie par `spy` qui permet de voir le squelette de matrice `sparse` et d'y repérer d'éventuelles régularités.

```
A=sprand(100,100,0.01) ; on crée une matrice creuse de taille 100 de densité 0,01
spy(A) on trace le squelette (voir Fig. 1.4)
```

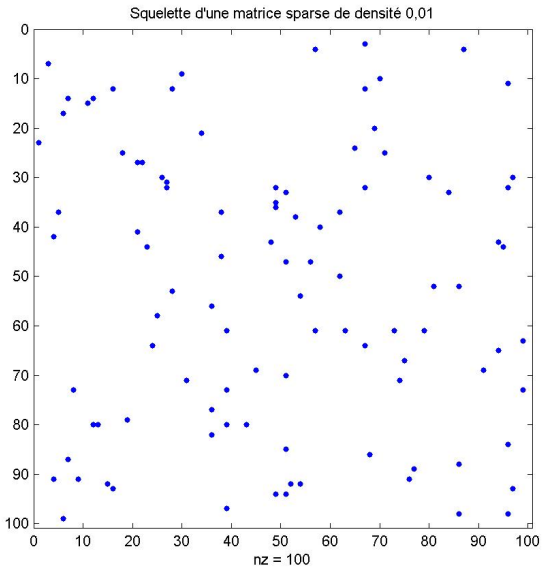


FIG. 2.5 – Exemple de matrice creuse

A titre d'exemple, traitons le cas de la matrice associée à l'équation de Laplace avec condition de Dirichlet homogène sur le carré unité lorsqu'on cherche à la résoudre par la méthodes des différences finies avec le schéma à 5 points, à pas égal dans les deux directions à δx. Pour n degrés de liberté,

on obtient la matrice de rigidité suivante :

$$A_n = \frac{1}{\delta x^2} \begin{pmatrix} 4 & -2 & 0 & \dots & 0 \\ -2 & 4 & -2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -2 & 4 & -2 \\ 0 & \dots & 0 & -2 & 4 \end{pmatrix}$$

On crée cette matrice `sparse` à l'aide de la commande `spdiags` par exemple pour $n = 50$:

```
e = ones(50,1);on crée un vecteur rempli de 1 qui va nous permettre de créer la matrice
A = spdiags([-2*e 4*e -2*e], -1 :1, 50, 50);on crée une matrice sparse de taille 50 dans laquelle on place les trois diagonales spécifiées par le premier argument aux emplacements indiqués par le deuxième argument
spy(A)
```

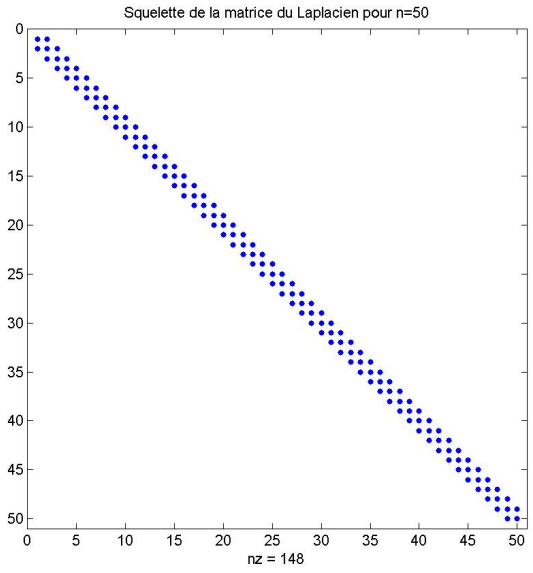


FIG. 2.6 – Squelette de la matrice du Laplacien discrétisé

On a donc présenté succintement les différentes fonctionnalités de Matlab propres à la fois à la géométrie et à la résolution des E.D.P. Il y a bien sûr d'autres matrices associées à un domaine, mais nous avons essayé de présenter les principales et en particulier celles que nous avons utilisées tout au long de ce projet. Pour bien tester *in vivo* ces fonctionnalités et la commande `assempde` de la PDE Toolbox, nous avons réalisé l'exemple suivant.

2.2 Exemple de résolution à l'aide de la PDE Toolbox sous Matlab

Soit D le disque unité. On s'intéresse au problème d'E.D.P suivant, on cherche u telle que :

$$\begin{cases} -\text{div}(a(x,y)\nabla u) = f & \text{dans } D \\ u = 0 & \text{sur } \partial D \end{cases}$$

Avec :

$$\begin{cases} a(x,y) = 2 + sin(xy) \\ f(x,y) = -4(2 + sin(xy) + xycos(xy)) \end{cases}$$

Ce problème admet une unique solution. On peut dans ce cas la calculer, il s'agit de :

$$u(x,y) = x^2 + y^2 + 1$$

On va donc pouvoir comparer la solution exacte et la solution approchée renvoyé par Matlab.

On a tout d'abord défini le disque en utilisant l'interface graphique de la PDE Toolbox de Matlab, et les conditions aux bords. Puis on exporte la géométrie, le maillage et les conditions aux bords, avec les matrices `b`, `p`, `e` et `t` précedemment décrites.

– On définit une fonction `grav` qui donne les coordonnées des centres de gravité des triangles.

```
function g=grav(p,t)
g=[(p(1,t(1,:))+p(1,t(2,:))+p(1,t(3,:)))/3;
(p(2,t(1,:))+p(2,t(2,:))+p(2,t(3,:)))/3];
```

– Voici la fonction Matlab qui donne f aux centres de gravité des triangles du maillage

```
function f=castestf(p,t)
s=grav(p,t);
x=s(1,:);
y=s(2,:);
f = -4*(2+sin(x.*y)+x.*y.*cos(x.*y));
```

x et y sont ici des vecteurs lignes contenant respectivement toutes les coordonnées sur (Ox) et (Oy) des centres de gravité

– Voici la fonction Matlab qui donne a aux centres de gravité des triangles du maillage.

```
function a=castesta(p,t)
s=grav(p,t);
x=s(1,:);
y=s(2,:);
a = 2+sin(x.*y);
```

– Voici la fonction Matlab qui donne u aux centres de gravité des triangles du maillage

```
function U=castestu(p,t)
s=grav(p,t);
x=s(1,:);
y=s(2,:);
U=x.^2+y.^2-1;
```

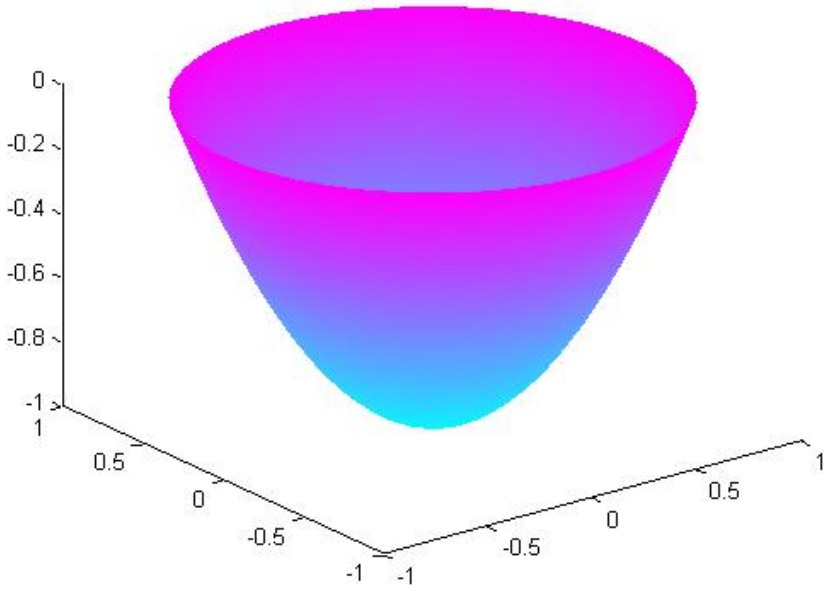


FIG. 2.7 – Tracé de la solution u exacte sur le disque unité

– On peut ensuite déterminer u de façon approchée en résolvant l'équation aux dérivées partielles, u sera donnée aux nœuds du maillage.

```
function Un=approxsol(b,p,e,t)
a=castesta(p,t);
On appelle a et f en tant que matfile
Un=assempde(b,p,e,t,a,0,f);
puis on utilise la fonction Matlab assempde qui résout le problème en utilisant les éléments finis.
La fonction assempde renvoie un vecteur U de taille m (nombre de points de maillage)
```

`assempde(b,p,e,t,a,c,f)` renvoie la solution approchée par la méthode des éléments finis de l'équation :

$$-div(a(x)\nabla x) + c(x)u(x) = f(x)$$

Dans le cas de cet exercice c vaut 0. Le vecteur obtenu est ensuite représenté graphiquement et

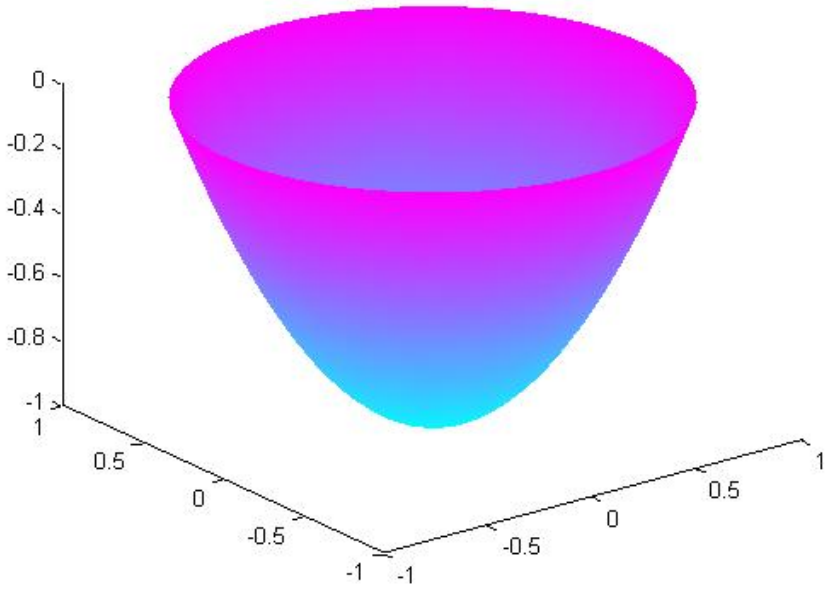


FIG. 2.8 – Tracé de la solution renvoyée par `assempde`

on a alors le graphe suivant :

– Finalement, le but de l'exercice étant de dessiner le graphe de la différence entre u calculé et u exact, on définit une fonction `delta` :

```
function d=delta(b,p,e,t)
un=approxsol(b,p,e,t);
valeurs de la solution aux nœuds du maillage

u=castestu(p,t);
valeurs aux centres de gravité des triangles

s=grav(p,t);
ug=griddata(s(1,:),s(2,:),u,p(1,:),p(2,:));
interpole u aux nœuds du maillage

d=un-ug(:);
pdesurf(p,t,d);
trace le graphe de d
```

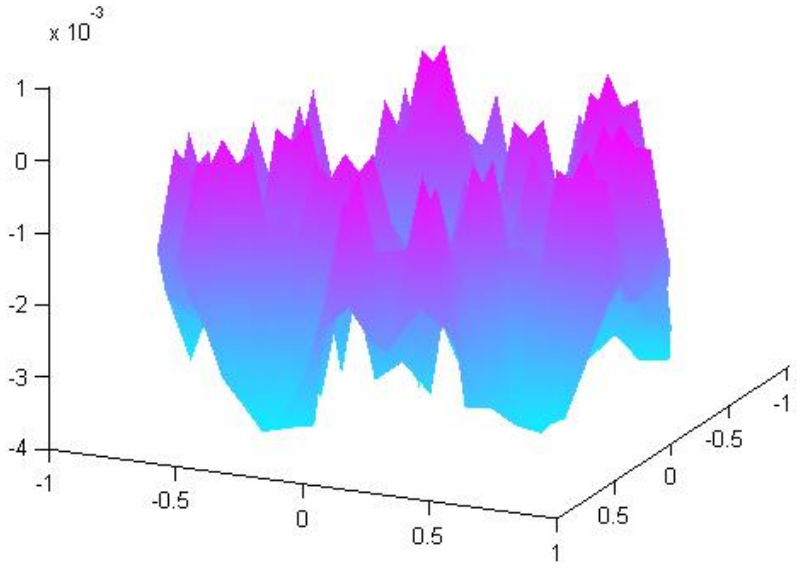


FIG. 2.9 – Tracé de l'erreur

Conclusion : On observe que la différence entre la solution exacte et la solution approchée ne dépasse pas 5.10^{-3} sur l'ensemble du disque unité. Ainsi, on peut conclure à la bonne approximation de la solution par la fonction Matlab.

Chapitre 3

Modélisation et résolution numérique

Cette section contient le coeur de notre projet. On y décrit dans la première partie la modélisation et la résolution théorique de notre problème grace à l'algorithme de Newton. Pour une complète compréhension de l'algorithme général, il est conseillé de se reporter au **chap. 5** qui présente l'algorithme de Newton. Dans la deuxième partie, on s'attache à l'implémentation de l'algorithme, ici encore, il est conseillé de se reporter au **chap. 5** pour une présentation de la méthodes des éléments finis. Enfin, dans la troisième partie, on implémente l'algorithme de Newton et on a recours à quelques variantes.

3.1 Modélisation simplifiée

Nous commencerons tout d'abord par résoudre le problème simplifié suivant

$$\begin{cases} -\text{div}(\sigma(T)\nabla V) = f & \text{dans } \Omega \\ -\text{div}(\kappa(T)\nabla T) = \sigma(T)\left|\nabla V\right|^2 & \text{dans } \Omega \\ V = T = 0 & \text{sur } \partial\Omega \end{cases}$$

avec :

$$\begin{array}{ll} \sigma(T) = \sigma_0 & \forall T \leq T_0 \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \geq T_c \end{array}$$

et :

$$\begin{array}{ll} \kappa(T) = L\sigma_0 T_0 & \forall T \leq T_0 \\ \kappa(T) = L\sigma(T)T = \frac{L\sigma_0 T}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \kappa(T) = L\sigma_c T_c & \forall T \geq T_c \end{array}$$

3.1.1 Résolution du problème théorique

Posons en premier lieu une nouvelle fonction $J(s) = \int_0^s \kappa(T)dT$. κ est continue de $\mathbb{R}^+ \to \mathbb{R}^+$ alors $\kappa(S) \geq k > 0$ et $J(s) \geq ks$. J est alors $\mathcal{C}^1$ de $\mathbb{R}^+ \to \mathbb{R}^+$ et strictement croissante. C'est donc une bijection et J^{-1} est aussi $\mathcal{C}^1$ de $\mathbb{R}^+ \to \mathbb{R}^+$.

Soit $u = J(T)$ alors $\nabla u = \kappa(T)\nabla T$ et on obtient le problème équivalent :

$$\begin{cases} -\text{div}(\gamma(u)\nabla V) = f & \text{dans } \Omega \\ -\Delta u = \gamma(u)\left|\nabla V\right|^2 & \text{dans } \Omega \\ V = u = 0 & \text{sur } \partial\Omega \end{cases}$$

avec $\gamma(u) = \sigma(J^{-1}(u))$

Mise en place des opérateurs

Pour notre problème, on cherche à résoudre un problème de point fixe $\xi(k) = k$ en utilisant l'algorithme de Newton. On introduit pour cela des opérateurs intermédiaires :

$$\xi_1 : u \longmapsto V \text{ solution de } \begin{cases} -\text{div}(\gamma(u)\nabla V) = f & \text{dans } \Omega \\ V = 0 & \text{sur } \partial\Omega \end{cases}$$

La définition de cet opérateur a bien un sens car la solution V du problème différentiel existe et est unique d'après le théorème de Lax-Milgram appliqué à $V \in H_0^1(\Omega)$ et en supposant que $f \in L^2(\Omega)$.

$$\tilde{\xi_2} : (X, Y) \longmapsto |\nabla Y|^2 \gamma(X)$$

$$\xi_2 : X \longmapsto \tilde{\xi_2}(X, \xi_1(X))$$

$$\xi_3 : g \longmapsto u \text{ solution de } \begin{cases} -\Delta u = g & \text{dans } \Omega \\ u = 0 & \text{sur } \partial\Omega \end{cases}$$

Cet opérateur est bien défini (comme ξ_1) dans $H_0^1(\Omega)$ pour $g \in H^{1/2}(\Omega)$. Alors $\xi = \xi_3 \circ \xi_2$ et on trouve la solution (u, V) du problème en cherchant le point fixe de $\xi(k) = k$.

3.1.2 Différentiation des opérateurs

Pour utiliser l'algorithme de Newton, nous devons différentier $\xi(k)$, car alors en posant $F(k) = \xi(k) - k$, on cherchera la solution δ de $F'(kn)(\delta) = -F(kn)$.
On a $\xi'(k)(h) = \xi_3'(\xi_2)\xi_2'(k)(h)$, calculons donc $\xi_3'(k)(h)$ et $\xi_2'(k)(h)$

Calcul de $\xi_1'(k)(h)$

$$-\text{div}(\gamma(u)\nabla V) = f \text{ dans } \Omega$$
$$V = 0 \text{ sur } \partial\Omega$$

Posons $a = \xi_1(l_o + h)$ et $b = \xi_1(l_o)$ La formulation variationnelle du problème différentiel donne :

$$\text{Pour } a : (1) \int_\Omega \gamma(l_o + h)\nabla a \nabla\varphi = \int_\Omega f\varphi$$

$$\text{Pour } b : (2) \int_\Omega \gamma(l_o)\nabla b \nabla\varphi = \int_\Omega f\varphi$$

γ est continue et dérivable sur $\mathbb{R}^+$ donc

$$\gamma(l_o + h) = \gamma(l_o) + h\gamma'(l_o) + ho(h)$$

$$\text{dans (1)} \int_\Omega \gamma(l_o)\nabla a\nabla\varphi + \int_\Omega h\gamma'(l_o)\nabla a\nabla\varphi + ho(h) = \int_\Omega f\varphi \text{ (1')}$$

$$\text{(1')-(2) donne} \int_\Omega \gamma(l_o)\nabla(a-b)\nabla\varphi = -\int_\Omega h\gamma'(l_o)\nabla a\nabla\varphi + ho(h)$$

Et ξ_1 continue donc $\nabla a = \nabla\xi_1(l_o + h) = \nabla\xi_1(l_o) + o(h)$
Alors

$$\xi_1'(l_o)(h) = a - b = \phi \text{ solution de } \int_\Omega \gamma(l_o)\nabla\phi\nabla\varphi = -\int_\Omega h\gamma'(l_o)\nabla\xi_1(l_o)\nabla\varphi + ho(h)$$

Calcul de $\xi_2'(k)(h)$ Commençons par différentier $\tilde{\xi_2}$:

$$\frac{\partial\tilde{\xi_2}}{\partial X}(X,Y)(hx) = \gamma'(X)\left|\nabla Y\right|^2 hx$$

$$\frac{\partial\tilde{\xi_2}}{\partial Y}(X,Y)(hy) = 2\gamma(X)\nabla Y\nabla hy$$

Ensuite, $\xi_2(X) = \tilde{\xi_2}(X,\xi_1(X))$, alors

$$\xi_2'(X)(h) = \frac{\partial\tilde{\xi_2}}{\partial X}(X,\xi_1(X))(h) + \frac{\partial\tilde{\xi_2}}{\partial Y}(X,\xi_1(X))\xi_1'(X)(h)$$

$$\xi_2'(X)(h) = \gamma'(X)\left|\nabla\xi_1(X)\right|^2 h + 2\gamma(X)\nabla\xi_1(X)\nabla\xi_1'(X)(h)$$

On a alors en posant $\xi_1(X) = \chi$

$$\Psi = \xi_2'(k)(h) = \gamma'(k)\left|\nabla\chi\right|^2 h + 2\gamma(k)\nabla\chi\nabla\phi$$

Calcul de $\xi_3'(k)(h)$

$$\xi_3 : g \longmapsto u \text{ solution de } \begin{cases} -\Delta u = g & \text{dans } \Omega \\ u = 0 & \text{sur } \partial\Omega \end{cases}$$

Cet opérateur est linéaire et continu donc

$$\xi_3'(k)(h) = \xi_3(h)$$

Calcul de $\xi'(k)(h)$

$$\xi'(k)(h) = \xi_3'(\xi_2)\xi_2'(k)(h)$$

$$\xi'(k)(h) = \xi_3(\xi_2'(k)(h))$$

Alors avec les notations

$$\begin{cases} \chi &= \xi_1(k) \\ \phi &= \xi_1'(k)(h) \\ \Psi &= \xi_2'(k)(h) = \gamma'(k)\left|\nabla\chi\right|^2 h + 2\gamma(k)\nabla\chi\nabla\phi \end{cases}$$

On a finalement

$$\xi'(k)(h) = u \text{ solution de } \begin{cases} -\Delta u = \Psi & \text{dans } \Omega \\ u = 0 & \text{sur } \partial\Omega \end{cases}$$

Mise en place de la résolution numérique

Algorithme de Newton Comme expliqué précédemment, on va appliquer l'algorithme de Newton pour résoudre ce problème de point fixe. En notant $F(k) = \xi(k) - k$, nous voulons déterminer k tel que $F(k) = 0$. On sait que $F'(k)(h) = \xi'(k)(h) - h$.
L'algorithme consistera alors à

$$\begin{cases} k_0 \text{ donné} \\ \text{chercher } \delta \text{ solution de } F'(k_n)(\delta) = -F(k_n) \\ k_{n+1} = k_n + \delta \end{cases}$$

Calculs numériques Avec les notations précédentes :

$$\begin{array}{ll} u = \xi_3(g) & u = C^{-1}G \\ \chi = \xi_1(k) & \chi = A^{-1}K \\ \phi = \xi_1'(k)(h) & \phi = A^{-1}Bh \\ \Psi = \xi_2'(k)(h) & \Psi = (E+F)h \end{array}$$

Avec

$$C_{ij} = \int_\Omega \nabla\varphi_i \nabla\varphi_j$$

$$A_{ij} = \int_\Omega \gamma(k)\nabla\varphi_i \nabla\varphi_j$$

$$B_{ij} = -\int_\Omega \gamma'(k)\varphi_j \nabla\varphi_i \nabla\chi$$

$$E_{ij} = \int_\Omega \gamma'(k)\left|\nabla\chi\right|^2 \varphi_i\varphi_j$$

$$D_{ij} = \int_\Omega 2\gamma(k)\nabla\chi\nabla\varphi_j\varphi_i$$

Alors $u = C^{-1}(DA^{-1}B + E)h$ et il faut résoudre

$$(C^{-1}(DA^{-1}B + E) - Id)h = k_n - \xi(k_n)$$

3.2 Mise en place algorithmique et résolution sous Matlab

Dans cette section, on décrit la démarche adoptée pour implémenter l'algorithme de Newton sous Matlab. On commence par présenter de manière générale les différentes étapes de la résolution, puis on examinera chacune des étapes en détail tout en commentant. On commence par entrer le domaine discrétisé Ω_h sous l'interface graphique :

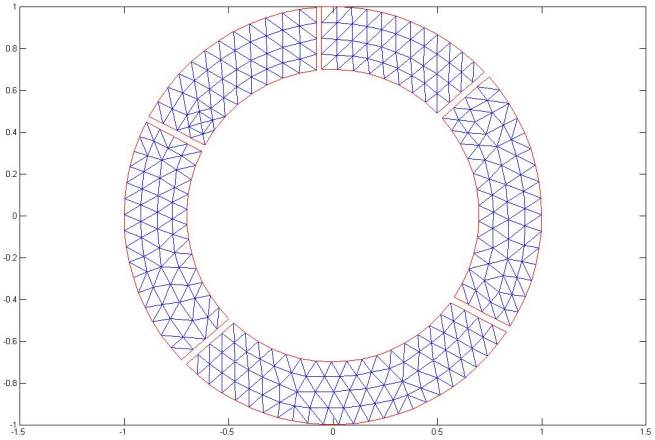


FIG. 3.1 – Domaine discrétisé Ω_h maillé une fois

3.2.1 Calcul de la fonction γ

On commence par calculer de manière approchée la fonction J définie par : $J(s) = \int_0^s \kappa(T)dT$. avec la fonction κ conductivité. Rappelons que les conductivités sont définies de la manière suivante :

$$\begin{array}{ll} \sigma(T) = \sigma_0 & \forall T \leq T_0 \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)} & \forall T \geq T_c \end{array}$$

et :

$$\begin{array}{ll} \kappa(T) = L\sigma_0 T_0 & \forall T \leq T_0 \\ \kappa(T) = L\sigma(T)T = \frac{L\sigma_0 T}{1+\alpha(T-T_0)} & \forall T \in [T_0, T_c] \\ \kappa(T) = L\sigma_c T_c & \forall T \geq T_c \end{array}$$

On commence donc par rentrer la fonction `sigma` qui prend pour argument un vecteur ou une matrice. Pour ce faire on choisit arbitrairement σ_0 et T_0 :

```
function sig=sigma(T)
```

Nom : sigma
Description : calcule la conductivité électrique pour un potentiel électrique, un vecteur ou une matrice de potentiels électriques donnés
Input : T un réel, un vecteur ou une matrice représentant le potentiel électrique en Volt
Output : sig la valeur de la conductivité électrique

```
L=1e-8;
To=3;
Tc=1e5;
so=2;
alpha=0.5;
sig=zeros(size(T));
for i=1:length(T)
if T(i)<=To
   sig(i)=so;
elseif (T(i)<Tc)
sig(i)=(so./(1+alpha.*(T(i)-To)));
else
    sig(i)=(so./(1+alpha.*(Tc-To)));
end
end
```

Puis on entre la fonction `kappa` :

```
function k=kappa(T)
```

Nom : kappa
Description : calcule la conductivité thermique pour une température, un vecteur ou une matrice de température donnés
Input : T un réel, un vecteur ou une matrice représentant la température en Kelvin
Output : k la valeur de la conductivité thermique

```
L=1e-8;
To=3;
Tc=1e5;
so=2;
alpha=0.5;
k=zeros(size(T));
for i=1:length(T)
if T(i)<=To
    k(i)=L.*so.*To;
elseif (T(i)<Tc)
k(i)=(L.*so.*T(i)./(1+alpha.*(T(i)-To)));
else
    k(i)=(L*so./(1+alpha.*(Tc-To))*Tc);
end
end
```

Une fois la fonction `kappa` créée, on crée une fonction `fJ` qui calcule J à l'aide de la commande `quadl` qui réalise l'intégration numérique de la fonction `kappa` à l'aide de la formule de quadrature de Lobatto.

```
function fctJ=fJ(U)
```

Nom : fJ
Description : donne une approximation numérique de l'intégrale de kappa
Input : y température et s valeur à un noeud
Output : fctJ valeur de l'intégrale pour un noeud du maillage

```
J=quadl(@kappa,0,y,1e-10);
fctJ=abs(J-s);
```

On calcule l'inverse de J à l'aide de la commande `fminsearch` qui permet de minimiser des fonctions numériques non linéaires et bornées, on se donne une tolérance de 10^{-10}

```
function finvJ=invJ(s)
```

Nom : invJ
Description : calcule la fonction réciproque de la fonction J par un vecteur aux noeuds du maillage
Input : s un vecteur de valeurs positives aux noeuds du maillage
Output : finvJ le vecteur des valeurs de la fonction réciproque J^{-1}

```
so=2;
L=1e-8;
finvJ=zeros(size(s));
for i=1:length(s)
finvJ=zeros(size(s));
    finvJ(i)=fminsearch(@fJ,0,optimset('TolFun',1e-10),s(i));
end
```

Remarque : on est obligé de boucler car `fminsearch` ne prend pas de vecteur comme argument. Cette fonction appelle beaucoup de calculs et est donc très couteuse.

Enfin on calcule la fonction γ qui va nous permettre le calcul des différents opérateurs par la suite, la fonction `gamma` prend pour argument un vecteur. Rappelons que γ est définie par : $\gamma(u) = \sigma(J^{-1}(u))$

```
function gam=gamma(X)
```

Nom : gamma
Description : calcule les valeurs de la fonction γ pour les noeuds du maillage
Input : X vecteur aux noeuds du maillage
Output : gam les valeurs de gamma pour ce vecteur X

```
gam=sigma(invJ(X));
```

Remarque : on appelle `invJ` qui utilise `fminbnd` et qui appelle `fJ` qui utilise `quadl`, cette opération est donc très couteuse. Pour un vecteur U de taille 1000, le calcul de $\gamma(U)$ nécessite plusieurs secondes de calculs (entre 4 et 5 secondes).

Avant de construire les opérateurs et les matrices associés à notre problème, on va créer deux fonctions `derivgamma` et `derivsigma` qui calculent les dérivées des fonctions `gamma` et `sigma`. Les dérivées des fonctions σ et γ valent :

$$\sigma'(T) = \frac{-\sigma_0 \alpha}{(1+\alpha(T-T_0))^2}$$

$$\gamma'(J^{-1}(u)) = \frac{-\kappa J^{-1}(u)\sigma_0 \alpha}{(1+\alpha(J^{-1}(u)-T_0))^2}$$

```
function gammaprime=derivgamma(invJ)
```

Nom : derivgamma
Description : calcule la dérivée de gamma pour un vecteur de réels positifs donné
Input : u un réel, un vecteur
Output : gammaprime le vecteur des dérivées de γ

```
invJprime=1./kappa(invJ);
gammaprime=derivsigma(invJ).*(invJprime);
```

```
function sig=derivsigma(T)
```

Nom : sigma
Description : calcule la dérivée de la conductivité électrique pour un potentiel électrique, un vecteur ou une matrice de potentiels
électriques donné
Input : T un réel, un vecteur ou une matrice représentant le potentiel électrique en Volt
Output : sig la valeur de la dérivée de la conductivité électrique

```
L=1e-8;
To=3;
Tc=1e5;
so=2;
alpha=0.5;
```

```
sig=zeros(size(T));
for i=1:length(T);
if (T(i)>=To)&(T(i)<Tc)
sig(i)=(-so*alpha./(1+alpha.*(T(i)-To)).^2);
else
    sig(i)=0;
end
end
```

3.2.2 Calcul des opérateurs et des matrices A,C et E

On va pouvoir calculer directement les opérateurs ξ_1,ξ_2 et ξ ainsi que les matrices A,C et E intervenant dans l'algorithme de Newton à l'aide de la commande `assempde` de la `pdetool` de Matlab. En effet, cette dernière commande renvoie la solution du problème aux équations différentielles mais peut aussi renvoyer différentes matrices associées au problème comme par exemple les matrices de masse et de raideur ou encore la matrice de correspondance noeuds/degrés de liberté (voir partie sur la commande `assempde`).

Calcul de ξ_1 rappelons que l'opérateur ξ_1 est défini de la manière suivante :

$$\xi_1 : u \longmapsto V \text{ solution de } \begin{cases} -\text{div}(\gamma(u_n)\nabla V) = f & \text{dans } \Omega \\ V = 0 & \text{sur } \partial\Omega \end{cases}$$

Il suffit donc d'appeler `assempde` avec comme argument la fonction f sur le bord et la fonction `gamma` appliqué au vecteur u_n On a donc la fonction suivante :

```
function ksi=ksi1(U,f,b,p,e,t,gam)
```

Nom : ksi1
Description : calcule la valeur de ksi1(U_n) aux centres de gravités des triangles du domaine
Input : Un vecteur des valeurs aux sommets des triangles du maillage vecteur ligne, f fonction à l'intérieur du domaine, on
peut la rentrer sous forme de string ou sous forme de vecteur, (b,p,e,t) matrices de codage du maillage et de la géométrie, gam vecteur $\gamma(U_n)$
Output : ksi vecteur des valeurs de ksi1 aux centres de gravités des triangles du maillage

```
gamma=pdeintrp(p,t,gam);
ksi=assempde(b,p,e,t,gamma,0,f);
```

Calcul de ξ_2 rappelons que l'opérateur ξ_2 est défini de la manière suivante :

$$\tilde{\xi_2} : (X, Y) \longmapsto |\nabla Y|^2 \gamma(X)$$

$$\xi_2 : X \longmapsto \tilde{\xi_2}(X, \xi_1(X))$$

On utilise cette fois la commande `pdegrad` de la `pdetool` qui renvoie le vecteur des valeurs du gradient de la solution aux barycentres du maillage et on rentre comme argument la fonction f, le vecteur $\gamma(u_n)$, la fonction ξ_1 et les matrices de codage du maillage.

```
function ksi=ksi2(ksi1,gam,b,p,e,t)
```

Nom : ksi2
Description : calcule les valeurs de ksi2 aux centres de gravités des triangles du maillage
Input : ksi1 vecteur des valeurs aux sommets des triangles du maillage vecteur ligne, f fonction à l'intérieur du domaine, on
peut la rentrer sous forme de string ou sous forme de vecteur, (b,p,e,t) matrices de codage du maillage et de la géométrie, gam vecteur $\gamma(U_n)$
Output : ksi vecteur des valeurs de ksi2 aux centres de gravités du maillage

```
[X1,Y1]=pdegrad(p,t,ksi1(U,f,b,p,e,t));
gamma=pdeintrp(p,t,gam);
```

on commence par calculer les composantes du gradient de ksi1

```
ksi=gamma.*(X1.^2+Y1.^2);
```

puis on calcule ksi2

Calcul de ξ rappelons que ξ est défini de la manière suivante :

$$\xi : g \longmapsto u \text{ solution de } \begin{cases} -\Delta u = g & \text{dans } \Omega \\ u = 0 & \text{sur } \partial\Omega \end{cases}$$

On utilise encore une fois la commande `assempde` avec cette fois la fonction ξ_2 sur le bord puis on définit l'opérateur F qui renvoie le second membre de l'équation de l'algorithme de Newton par :

$$F(u_n) = \xi(u_n) - u_n$$

Et on le programme suivant :

```
function F1=SecMembre(U,ksi2,f,b,p,e,t)
```

Nom : SecMembre
Description : calcule le second membre de l'équation de Newton pour un
vecteur aux noeuds du maillage
Input : U vecteur des valeurs aux noeuds du maillage vecteur ligne, ksi2 vecteur $\xi_2(U)$ f fonction à l'intérieur du domaine, on
peut la rentrer sous forme de string ou sous forme de vecteur, (b,p,e,t) matrices de codage du maillage et de la géométrie
Output : F1 vecteur des valeurs du second membre aux noeuds du maillage on commence par calculer psi2 aux noeuds du maillage

```
ksi=assempde(b,p,e,t,1,0,ks2);
```

on entre ksi2 comme argument dans assempde pour calculer ksi

```
F1=ksi-U;
```

Le calcul des matrices A, C et E se fait lui aussi à l'aide d'`assempde` et on a les programmes suivants :

Calcul de A

```
function A=matriceA(gam,b,p,e,t)
```

Nom : matriceA
*Description : calcule la matrice A à l'aide d'**assempde** pour un vecteur aux noeuds du maillage*
Input : gam vecteur $\gamma(Un)$ aux noeuds du maillage vecteur ligne,(b,p,e,t) matrices de codage du maillage et de la géométrie
Output : A matrice de rigidité du système (change à chaque itération de l'algorithme de Newton), de taille N : nombre de noeuds du maillage,
Matlab la renvoie directement sous forme sparse

```
gamma=pdeintrp(p,t,gam);
[K,F,B,ud]=assempde(b,p,e,t,gamma,0,0);
A=K;
```

La matrice ne dépend que du vecteur $\gamma(u_n)$ retourné à chaque itération de l'algorithme de Newton.

Calcul de C

```
function [C1,Bconnect]=matriceC(b,p,e,t);
```

Nom : matriceC
*Description : calcule la matrice de rigidité C à l'aide d'**assempde***
Input : (b,p,e,t) matrices de codage du maillage et de la géométrie
Output : C1 matrice de rigidité du système (elle reste constante tout au long de l'algorithme), de taille N : nombre de noeuds du maillage, Bconnect matrice de correspondance ddl/noeuds
Matlab les renvoie directement sous forme sparse

```
[K,F,Bconnect,ud]=assempde(b,p,e,t,1,0,0);
C1=K;
```

La matrice C ne varie pas au cours de la résolution, il suffit donc de la calculer une fois puis de la rentrer comme argument des fonctions ultérieures, de plus, on en profite pour renvoyer une matrice `Bconnect` qui reste aussi constante tout au long de l'algorithme, la matrice de correspondance degrés de libertés/noeuds (voir partie suivante).

Calcul de E

```
function E=matriceE(U,ks1,f,b,p,e,t,Bconnect)
```

Nom : matriceE
*Description : calcule la matrice E à l'aide d'**assempde***
Input : U et ks1 vecteur $\xi(U_n)$ aux centres de gravité, f, (b,p,e,t) matrices de codage du maillage et de la géométrie,
Bconnect matrice de correspondance ddl/noeuds
Output : E matrice du système (elle change à chaque itération)
de taille N : nombre de noeuds du maillage,
Matlab les renvoie directement sous forme sparse

```
Un=pdeintrp(p,t,U);
```

retourne aux noeuds

```
[px,py]=pdegrad(p,t,ks1);
```

aux centres de gravité

```
[K,u,F,Q,G,H,R]=assempde(b,p,e,t,0,derivsigma(Un).*(px.^2+py.^2),0);
E1=u;
```

aux noeuds

```
E=Bconnect'*E1*Bconnect;
```

aux degrés de liberté

Ici la matrice E dépend à la fois de l'étape de l'algorithme et de la fonction sur le bord. E doit donc être calculée à chaque étape.

3.2.3 Calcul des matrices B et D

Le calcul des matrice B et D est plus délicat que les matrices précédentes car Matlab ne renvoie pas directement ces matrices. L'enjeu de l'algorithme est donc de remplir correctement ces matrices. Il s'agit d'une part de définir la connectivité du maillage i.e les couples de noeuds qui sont connectés l'un à l'autre et les triangles associés et d'autre part de trouver des formules d'approximation pour le calcul des coefficients des matrices B et D.

Connectivité du maillage En premier lieu, remarquons que toutes les matrices qui interviennent dans l'algorithme ont toute la même connectivité que la matrice de masse C. En effet, la matrice C a pour coefficient général :

$$C_{ij} = \int_\Omega \nabla\phi_i \nabla\phi_j$$

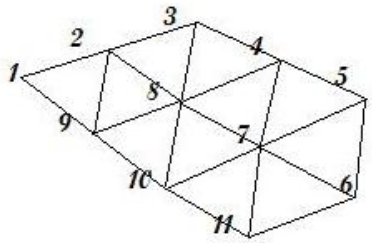
Ainsi, deux noeuds i et j sont connectés si et seulement si le coefficient C_ij est non nul. On utilise alors à profit le format `sparse` pour obtenir les vecteurs qui contiennent les coordonnées de éléments non nuls autrement dit les couples des noeuds connectés, à l'aide de la commande suivante :

```
[ic,jc,sc]=find(C);
```

Les vecteurs `ic` et `jc` contiennent alors les numéros de lignes et de colonnes des éléments non nuls, rangés dans l'ordre de lecture de Matlab (voir le chapitre **Quelques précisions sur Matlab**). Ces deux vecteurs vont nous permettre de parcourir l'ensemble des noeuds connectés et seulement ceux-là. Il est à noter que les trois vecteurs `ic`, `jc` et `sc` ont tous tous les trois la même taille α le nombre d'éléments non nuls de la matrice C.

Noeuds intérieurs et degrés de liberté Avant d'utiliser la connectivité du maillage pour construire les matrices B et D, il convient de distinguer deux types de noeuds du maillages : les noeuds intérieurs et les noeuds sur le bord. Notons n_m le nombre de noeuds du maillage et n_i le nombre de noeuds à l'intérieur du maillage. Dans notre modélisation, on a pris des conditions de Dirichlet homogènes, la valeur de la solution sur le bord est donc déjà connue et vaut 0. Il est donc inutile de traiter les valeurs de la solution sur le bord comme des inconnues (de même, dans le cas de conditions de Dirichlet inhomogènes la valeur est aussi connue sur le bord). Les véritables valeurs

pour lesquelles on cherche la solution sont celles des noeuds intérieurs et on parle de *degrés de liberté*. Le nombre d'inconnues passe alors de n_m à n_i, de même que la taille des matrices associées. Cette diminution n'est pas négligeable au vu de la taille des systèmes à résoudre. A titre d'exemple, dans le cas d'un maillage raffiné une fois de la couronne effectué sur la PDE Toolbox, on passe de $n_m = 361$ noeuds du maillage à $n_i = 185$ degrés de liberté, la taille des systèmes à résoudre est diminue de moitié ce qui est profitable d'un point de vue coût en calcul. Il nous faut donc trouver une procédure simple qui permette de passer des matrices et des vecteurs pour l'ensemble du maillage aux matrices et vecteurs pour les degrés de liberté. En effet, quand on supprime les noeuds au bord, on diminue la taille de la matrice mais on change aussi la numérotation. Une manière de surmonter ce problème est fournie par la commande `assempde` qui renvoie une matrice de correspondance noeuds/degrés de liberté, notée P. Cette matrice est de taille $n_m * n_i$, la ligne i_1 pour i_1 noeud sur le bord ne contient que des zéros et la ligne i_2 pour i_2 degré de liberté contient un 1 dans la colonne k, où k est le numéro de i_2 dans la nouvelle numérotation. l s'agit d'une matrice particulièrement creuse et en supprimant toutes les lignes correspondantes au noeuds sur le bord, on obtient la matrice identité. Par exemple, dans le cas particulièrement simple suivant :
On obtient la matrice P suivante de $\mathcal{M}_{11,2}(\mathbb{R})$:



$$P = \begin{pmatrix} 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Et on passe alors très facilement d'un vecteur ou d'une matrice aux noeuds du maillage au même vecteur ou la même matrice mais seulement sur les degrés de liberté, de la manière suivante : si $u \in \mathbb{R}^n_m$ est un vecteur des valeurs de u sur tous les noeuds, alors $P^T u \in \mathbb{R}^n_i$ est le vecteur des valeurs de u aux degrés de liberté, et si $C \in \mathcal{M}_{n_m}(\mathbb{R})$ est une matrice aux noeuds du maillage, alors $P^T C P \in \mathcal{M}_{n_i}(\mathbb{R})$ est cette même matrice aux degrés de liberté. Enfin, on obtient cette matrice P directement sous Matlab avec la commande :

```
[C,F,P,ud]=assempde(b,p,e,t,c,a,f);
```

La matrice `P` est alors la matrice de correspondance.

Assemblage des matrices B et D Pour commencer, on crée une fonction qui renvoie les numéros des triangles associés à un couple (i, j) (ou i peut être égal à j) :

```
function v=triang(i,j,t)
```

Nom : triang
Description : renvoie la liste de triangles qui contiennent les sommets i et j
Input : numéros i et j de deux sommets du maillage, t matrice des triangles du maillage t
Output : v vecteur de longueur variable

```
a=[find(t(1,:)==i) find(t(2,:)==i) find(t(3,:)==i)];
v=a([find(t(1,a)==j) find(t(2,a)==j) find(t(3,a)==j)]);
```

Puis une fonction qui stocke tous ces numéros dans une matrice.

```
function tr=listtriangl(C,Bconnect,p,t)
```

Nom : listtriangl
Description : renvoie la table qui contient la liste des triangles connectés
à chaque couple (i,j) de noeuds
Input : matrice de masse C, matrice de correspondance Bconnect, p et t matrices des points et des triangles
du maillage
*Output : tr matrice de taille Nombre de connections * 10*

```
[i1,j1,k1]=find(C);
[nn,ni]=size(Bconnect);
e=[1:nn]';
e2=[1:ni]';
ddl2N=Bconnect'*e;
N2ddl=Bconnect*e2;
x=length(i1);
tr=zeros(x,10);
for i=1:x
    in=ddl2N(i1(i));
    jn=ddl2N(j1(i));
    n=triang(in,jn,t);
    tr(i,:)=[n,zeros(1,10-length(n))];
end
```

Remarque : le nombre de colonnes de `tr` est de 10 car on a supposé qu'un noeud n'était jamais connecté à plus de 10 triangles distincts.

Rappelons que les matrices B et D sont définies de la manière suivante :

$$B_{ij} = -\int_\Omega \gamma'(k)\varphi_j \nabla\varphi_i \nabla\chi$$

$$D_{ij} = \int_\Omega 2\gamma(k)\nabla\chi\nabla\varphi_j\varphi_i$$

Pour calculer les coefficients B_{ij} et D_{ij} où les indices ij désigne les degrés de liberté i et j, on a besoin de savoir à quels triangles les sommets i et j appartiennent (i.e l'indice de colonne de la matrice `t` renvoyée par la `pdetool` dans laquelle figurent i et j) et les degrés de liberté auquels les noeuds i et j sont connectés. Une fois ces données acquises, on pourra appliquer les formules d'approximation pour

chaque couple de noeuds et ainsi construire les matrices. De plus pour pouvoir calculer les gradients sur un triangle il faut savoir dans quel ordre sont rangés les sommets du triangle dans la matrice `t`, c'est l'objet de la fonction `position` :

```
function [posi,posj]=position(i1,j1,t,numt)

Nom : position
Description : renvoie les positions relatives dans la matrice t des
degrés de liberté i1 et j1
Input : i1,j1 deux degrés de liberté, t matrice des triangles, numt
numéro du triangle qui contient i1 et j1 dans la matrice t
Output : les positions de i1 et j1 (1,2 ou 3)

t1=t(1:3,:);
posi=find(~(t(1:3,numt)-i1));
posj=find(~(t(1:3,numt)-j1));
```

On a les formules d'approximation suivantes. Pour tout couple (i, j) de degrés de liberté connectés entre eux, les coefficients des matrices B et D sont les suivants (où l'on somme sur les triangles connectés à ce couple) :

$$B_{ij} = \sum \nabla\phi_i(P_k)\gamma'(u_{Nk})\nabla\chi(u_{Nk})\frac{|T_k|}{3}$$

$$D_{ij} = \sum \nabla\phi_j(P_k)\gamma(u_{Nk})\nabla\chi(u_{Nk})\frac{2|T_k|}{3}$$

où :
- N_k désigne les numéros des triangles connectés au couple (i, j)
- $|T_k|$ désigne les aires de ces triangles
- P_k désigne les barycentres de ces triangles

Pour calculer ces quantités, on utilise la commande `pdetrg` qui renvoie la valeur du gradient des fonctions "chapeaux" ϕ_i et finalement on a le programme suivant `MatBD` qui renvoie les deux matrices B et D :

```
function [B,D]=MatBD(U,ksi1,gamma,derivgamma,f,b,p,e,t,tr,Bconnect,C1)
```

Nom : MatBD
Description : construit les matrices B et D aux degrés de liberté pour
un vecteur U aux noeuds du maillage
Input : le vecteur U aux noeuds calculé à chaque itération de l'algorithme de Newton
les vecteurs $\xi(U)$, $\gamma(U)$ et $\gamma'(U)$, la fonction f, (b,p,e,t) les matrices de codage, tr liste des triangles,
Bconnect
C1 matrice de masse
Output : les matrices B et D

```
C=C1;
[i,j,S]=find(C);
[ar,g1x,g1y,g2x,g2y,g3x,g3y]=pdetrg(p,t);
S=zeros(length(i),1);
B=sparse(i,j,S);
D=sparse(i,j,S);
tgrad=zeros(6,length(ar));
tgrad(1,:)=g1x;
tgrad(2,:)=g1y;
tgrad(3,:)=g2x;
tgrad(4,:)=g2y;
tgrad(5,:)=g3x;
tgrad(6,:)=g3y;
[nn,ni]=size(Bconnect);
e=[1:nn]';
e2=[1:ni]';
ddl2N=Bconnect'*e;
N2ddl=Bconnect*e2;
```

on commence par calculer les vecteurs des aires des triangles et des
valeurs du gradient en chacun des points de chaque triangle

```
[px,py]=pdegrad(p,t,ksi1);
G=pdeintrp(p,t,gamma)';
DG=pdeintrp(p,t,derivgamma)';
```

on calcule le gradient de ξ_1

```
for k=1:length(i)
    ddli=i(k);
    ddlj=j(k);
    i1=ddl2N(ddli);
    j1=ddl2N(ddlj);
    v=nonzeros(tr(k,:));
            for x=1:length(v)
                numt=v(x);
                [posi,posj]=position(i1,j1,t,numt);
                B(ddli,ddlj)=B(ddli,ddlj)-(ar(numt)*DG(numt)/3)*(px(numt)*
                tgrad(2*posi-1,numt) + py(numt)*tgrad(2*posi,numt));
```

```
                D(ddli,ddlj)=D(ddli,ddlj)+(2*ar(numt)*G(numt)/3)*(px(numt)*
                tgrad(2*posj-1,numt) + py(numt)*tgrad(2*posj,numt));
        end
    end
```

On utilise donc à profit le format `sparse` en ne parcourant que les vecteurs des indices des coefficients non nuls de la matrice C qui correspondent bien aux noeuds connectés, ceci nous permet de réduire considérablement les boucles. Nous avons néanmoins cherché à programmer le remplissage des matrices de manière vectorielle mais les structures de données différentes rendait la chose plus complexe que des boucles.

3.3 Algorithme de Newton et variantes

3.3.1 Méthode directe

Maintenant qu'on a assemblé les matrices correctement, on peut construire la matrice d'itération du système qu'il faut résoudre à chaque itération. Rappelons que cette matrice est définie par :

$$C^{-1}(D(u_n, f)A(u_n)^{-1}B(u_n, f) + E(u_n, f)) - I_n$$

et que le système à résoudre à chaque étape est :

$$(C^{-1}(D(u_n, f)A(u_n)^{-1}B(u_n, f) + E(u_n, f)) - I_n)h = u_n - \xi(u_n) \quad \text{où } h \text{ est l'inconnue}$$

Il s'agit d'un système de très grande taille, on ne va donc pas inverser la matrice pour le résoudre, mais au contraire chercher différentes méthodes directes ou indirectes (itératives type gradient conjugué) pour le résoudre. Dans un premier temps, on programme l'algorithme de Newton avec l'opérateur `backslash` de Matlab qui permet la résolution de systèmes linéaires de manière directe de manière efficace, i.e en utilisant la réduction de Gauss avec pivot total qui est de complexité $\mathcal{O}(n^3)$:

```
function [Res,Sol,V,T,err]=Resoldirecte(U,Imax,f,b,p,e,t,C1,tr,Bconnect)
```

Nom : Resoldirecte
Description : algorithme de Newton avec résolution directe du système à chaque étape
Input : U vecteur de départ aux noeuds (on prend garde à partir avec des valeurs nuls aux noeuds du bord), Imax nombre d'itérations maximum, f fonction sur le bord, (b,p,e,t) matrices de codage du maillage, C1 matrice de masse, tr liste des triangles, Bconnect matrice de correspondance ddl/noeuds
Output : Res valeur du résidu (norme2 entre deux vecteurs successifs), Sol vecteur de la solution approchée aux noeuds, V et T le température et le potentiel sur la couronne err le vecteur des erreurs à chaque étape,trace le vecteur initiale et la solution approchée sur deux graphiques séparés

```
i=0;
close all;
pdesurf(p,t,U)
```

```
tol=1e-8;
erreur=tol+1;
[v,v]=size(C1);
err=zeros(1,Imax);
while(erreur > tol)&(i<Imax)
    finvJ=invJ(U);
    gam=gamma(finvJ);
    ksi=ksi1(f,b,p,e,t,gam);
    ks=ksi2(ksi,gam,f,b,p,e,t);
    gammaprime=derivgamma(finvJ);
    S=-SecMembre(U,ks,f,b,p,e,t)'*Bconnect;

    on enlève les noeuds extérieurs

    A1=matriceA(gam,b,p,e,t);
    E1=matriceE(U,ksi,f,b,p,e,t,Bconnect);
    [B1,D1]=MatBD(U,ksi,gam,gammaprime,f,b,p,e,t,tr,Bconnect,C1);
    h=(aux*(D1*inv(A1)*B1+E1)-speye(v))\S';
    h2=Bconnect*h;


    on résout directement et on remet aux noeuds

    U=U+h2;
    i=i+1;
    erreur=norm(h)/norm(U)
    err(i)=erreur;
end
i
Sol=U;
Res=err(i);
figure;
pdesurf(p,t,Sol);
T=invJ(Sol);
figure;
pdesurf(p,t,T);
sig=pdeintrp(p,t,sigma(T));
V=assempde(b,p,e,t,sig,0,f);
figure;
pdesurf(p,t,V);
```

3.3.2 Méthode GMRES

Mais on peut utiliser d'autres méthodes, en particulier pour la résolution du système à chaque étape. Dans le cas test, on a par exemple utilisé une méthode de résolution indirecte. Dans le cas de très grands systèmes, on utilise souvent la méthode GMRES (*Generalized Minimum Residual Method*) qui est une version améliorée de l'algorithme du gradient conjugué qu'on applique à des fonctionnelles non symétriques (cf. **Chap. 5**). Pour ce faire, on a besoin d'une fonction `prodmat` qui réalise à moindre coût le produit matrice-vecteur du premier membre du système linéaire en utilisant la factorisation de Choleski de la matrice C :

```
function y=prodmat(x,R,D,A,B,E)
```

Nom : prod
Description : réalise le produit matrice vecteur du premier membre du système linéaire à résoudre à chaque étape, de manière moins couteuse et pour pouvoir appliquer la méthode GMRES
Input : x vecteur aux degrés de liberté, R factorisée de Choleski de la matrice C, D, A, B, E les matrices construites à chaque étape
Output : produit y=(inv(C)(D*inv(A)*B+E) - In)x*

```
aux=D*inv(A)*B+E;
y=R\(R'\(aux*x))-x;
```

Puis on appelle la commande `GMRES` de Matlab dans l'algorithme de Newton :

```
function [Res,Sol,V,T,err]=ResolGMRES(U,Imax,f,b,p,e,t,C1,tr,Bconnect)
```

Nom : ResolGMRES
Description : algorithme de Newton avec résolution GMRES du système à chaque étape
Input : U vecteur de départ aux noeuds (on prend garde à partir avec des valeurs nuls aux noeuds du bord), Imax nombre d'itérations maximum, f fonction sur le bord, (b,p,e,t) matrices de codage du maillage, C1 matrice de masse, tr liste des triangles, Bconnect matrice de correspondance ddl/noeuds
Output : Res valeur du résidu (norme2 entre deux vecteurs successifs), Sol vecteur de la solution approchée aux noeuds, V et T le température et le potentiel sur la couronne err le vecteur des erreurs à chaque étape,trace le vecteur initiale et la solution approchée sur deux graphiques séparés

```
i=0;
close all;
pdesurf(p,t,U)
tol=1e-8;
erreur=tol+1;
[v,v]=size(C1);
err=zeros(1,Imax);
while(erreur > tol)&(i<Imax)
    finvJ=invJ(U);
    gam=gamma(finvJ);
    ksi=ksi1(f,b,p,e,t,gam);
    ks=ksi2(ksi,gam,f,b,p,e,t);
    gammaprime=derivgamma(finvJ);
    S=-SecMembre(U,ks,f,b,p,e,t)'*Bconnect;

    on enlève les noeuds extérieurs

A1=matriceA(gam,b,p,e,t);
    E1=matriceE(U,ksi,f,b,p,e,t,Bconnect);
    [B1,D1]=MatBD(U,ksi,gam,gammaprime,f,b,p,e,t,tr,Bconnect,C1);
    n=0;
```

```
    h=zeros(v,1);
    [h,flag,res]=gmres('prodmat',S',100,1e-8,10,[],[],zeros(v,1),R,D1,A1,B1,E1);
    res
    h2=Bconnect*h;

    on résout à l'aide de GMRES, on appelle le résidu de la méthode
pour contrôler la convergence de la méthode et on remet aux noeuds

    U=U+h2;
    i=i+1;
    erreur=norm(h)/norm(U)
    err(i)=erreur;
end
i
Sol=U;
Res=err(i);
figure;
pdesurf(p,t,Sol);
T=invJ(Sol);
figure;
pdesurf(p,t,T);
sig=pdeintrp(p,t,sigma(T));
V=assempde(b,p,e,t,sig,0,f);
figure;
pdesurf(p,t,V);
```

3.3.3 Newton à pas optimal

Enfin, on a programmé la méthode de Newton à pas optimal qui en général converge en moins d'itérations que la résolution directe au prix d'une augmentation du temps de calcul (cf. **Chap. 5**). On commence par créer une fonction `optimal` qui évalue la norme du second membre du système "modifié" par le pas s :

```
function fct_opt=optimal(s,U,f,ks2,h,b,p,e,t,Bconnect)

Nom : optimal
Description : évalue la norme du second membre construit à l'aide
du pas s, on va l'utiliser avec fminsearch
Input : s le pas (entre 0 et 1), U vecteur aux ddl, f, vecteur ks2(U),
h solution du système de Newton, (b,p,e,t) matrices de codage, Bconnect
matrice de correspondance
Output : norme du second membre modifié du pas s

F1=SecMembre(U+s*h,ks2,f,b,p,e,t)'*Bconnect;
fct_opt=norm(F1);
```

Puis, on appelle `fminsearch` dans l'algorithme de Newton :

```
function [Res,Sol,V,T,err]=Resoloptimal(U,Imax,f,b,p,e,t,C1,tr,Bconnect)
```

Nom : Resoloptimal
Description : algorithme de Newton à pas optimal
Input : U vecteur de départ aux noeuds (on prend garde à partir avec des valeurs nuls aux noeuds du bord), Imax nombre d'itérations maximum, f fonction sur le bord, (b,p,e,t) matrices de codage du maillage, C1 matrice de masse, tr liste des triangles, Bconnect matrice de correspondance ddl/noeuds
Output : Res valeur du résidu (norme2 entre deux vecteurs successifs), Sol vecteur de la solution approchée aux noeuds, V et T le température et le potentiel sur la couronne err le vecteur des erreurs à chaque étape,trace le vecteur initiale et la solution approchée sur deux graphiques séparés

```
i=0;
close all;
pdesurf(p,t,U)
tol=1e-8;
erreur=tol+1;
[v,v]=size(C1);
err=zeros(1,Imax);
while(erreur > tol)&(i<Imax)
    finvJ=invJ(U);
    gam=gamma(finvJ);
    ksi=ksi1(f,b,p,e,t,gam);
    ks=ksi2(ksi,gam,f,b,p,e,t);
    gammaprime=derivgamma(finvJ);
    S=-SecMembre(U,ks,f,b,p,e,t)'*Bconnect;

    on enlève les noeuds extérieurs

    A1=matriceA(gam,b,p,e,t);
    E1=matriceE(U,ksi,f,b,p,e,t,Bconnect);
    [B1,D1]=MatBD(U,ksi,gam,gammaprime,f,b,p,e,t,tr,Bconnect,C1);
    h=(aux*(D1*inv(A1)*B1+E1)-speye(v))\S';
    h2=Bconnect*h;
    rho=fminbnd(@optimal,0,1,[],U,f,ks,h2,b,p,e,t,Bconnect);
    U=U+rho*h2;


    on résout directement le système,on cherche le pas optimal et on construit l'itéré
en multipliant par ce pas optimal. On remet aux noeuds

    U=U+h2;
    i=i+1;
    erreur=norm(h)/norm(U)
    err(i)=erreur;
end
i
Sol=U;
Res=err(i);
figure;
```

```
pdesurf(p,t,Sol);
T=invJ(Sol);
figure;
pdesurf(p,t,T);
sig=pdeintrp(p,t,sigma(T));
V=assempde(b,p,e,t,sig,0,f);
figure;
pdesurf(p,t,V);
```

Il nous reste maintenant à tester nos algorithmes sur un cas test. C'est-à-dire, comme dans l'exemple du **Chap. 2**, comparer la solution exacte avec la solution approchée renvoyée par notre algorithme.

Chapitre 4

Mise en place d'un cas test pour notre algorithme

Afin de tester notre algorithme, et de déterminer si nous pouvons l'appliquer à un cas réel, nous avons décidé de programmer un cas test qui répondra à ces questions.

Pour cela, nous avons décidé de construire une solution exacte du problème et de la comparer à la solution donnée par l'algorithme.

4.1 Enoncé du nouveau problème

Commençons par alléger le problème de départ afin de pouvoir déterminer une solution exacte.
Le problème initial qui nous a été posé était :

$$\begin{cases} -\text{div}(\sigma(T)\nabla V) = f & \text{dans } \Omega \\ -\text{div}(\kappa(T)\nabla T) = \sigma(T)\left|\nabla V\right|^2 & \text{dans } \Omega \\ V = T = 0 & \text{sur } \partial\Omega \end{cases}$$

avec $\kappa(T) = L\sigma(T)T, L \in \mathbb{R}$ et $\sigma(T) = \frac{\sigma_0}{1+\alpha(T-T_0)}, \sigma_0, \alpha \in \mathbb{R}$

Prenons par exemple pour notre cas test
$\kappa(T) = 1$ et $\sigma(T) = T^2 + 1$

On impose de plus un nouveau domaine : $\Omega = [0,1] \times [0,1]$
Et on connait la solution $V = sin(\pi x)sin(\pi y)$

De là, on va chercher T solution de

$$\begin{cases} -\Delta T = \sigma(T)\left|\nabla V\right|^2 & \text{dans } \Omega \\ T = 0 & \text{sur } \partial\Omega \end{cases}$$

Une fois T trouvé, on prendra $f = -\text{div}(\sigma(T)\nabla V)$.

Nous pourrons conclure en imposant ce $\sigma(T)$, ce $\kappa(T)$, et ce f dans notre algorithme de Newton, pour avoir la solution (V, T) correspondante, il nous suffira alors de les comparer.

4.2 Résolution de ce nouveau problème

La question est donc de résoudre

$$\begin{cases} -\Delta T = \sigma(T)\,|\nabla V|^2 & \text{dans } \Omega \\ T = 0 & \text{sur } \partial\Omega \end{cases}$$

de manière la plus exacte possible.

Cette équation différentielle étant non linéaire, la résolution numérique semble être le moyen le plus efficace d'aborder le problème.

Cependant, dans un souci d'exactitude, nous sommes obligés d'utiliser un maillage très fin et régulier, appelé *maillage de Poisson*, dans lequel nous ferons appel à la fonction Matlab pdenonlin. Le maillage de Poisson s'applique sur des géométries très simples type carré ou rectangle et `pdenonlin` utilise la transformée de Fourier. La résolution est plus précise qu'avec un maillage type Delaunay. Voici un exemple de maillage de Poisson :

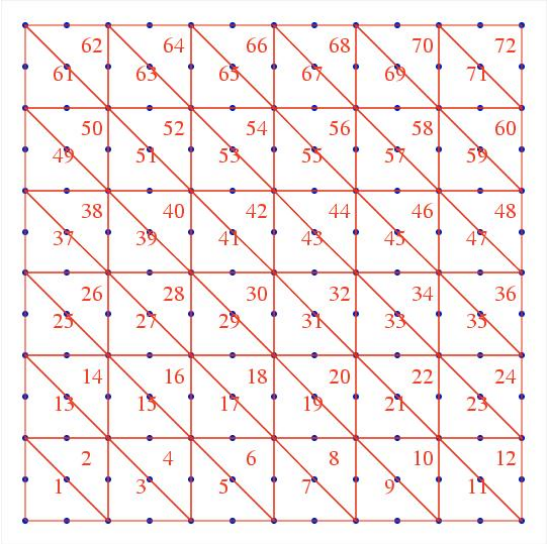


FIG. 4.1 – Maillage de Poisson

Il faut d'abord définir ce maillage, pour cela on doit rentrer le code Matlab :

```
[p1,e1,t1]=poimesh(g,20)
```

qui divisera le rectangle avec un pas de $\frac{1}{20}$. On va également créer un maillage plus grossier sur lequel on interpolera nos résultats finaux dans le but de les comparer avec ceux de l'algorithme qui lui se fera sur le maillage moins fin, pour réduire les temps de calculs :

```
[p,e,t]=poimesh(g,10)
```

On enregistre toutes ces données dans un fichier qu'on nomme `donnee2`, ainsi il suffira de charger (`load donnee2`) au début de chaque fonction, au lieu de tout mettre en argument.

On va ensuite définir numériquement les deux fonctions donnant respectivement σ et sa dérivée :

```
function sig=sigmatest(T)
sig=(T.^2+1);

function sigmaprime=derivsigmatest(T)
sigmaprime=2*T;
```

On définit ensuite notre fonction V numériquement, le laplacien de V dont on se servira plus tard :

```
function [V,laplacienV]=Vcastest
load donnee2 p1 t1;
x=p1(1,:);
y=p1(2,:);
V=sin(pi*x).*sin(pi*y);

laplacienV=-2*pi^2*sin(pi*y).*sin(pi*x);
```

Il faut à présent résoudre le problème différentiel non linéaire afin de trouver T :

```
function T=Tcastest
load donnee2 b p t p1 e1 t1;

[u,res]=pdenonlin(b,p1,e1,t1,1,0,'((cos(pi*x).*sin(pi*y)).^2+(cos(pi*y).*sin(pi*x))
.^2).*((pi^2)*(u.^2+1))');
T=u;
```

On en déduit finalement f :

```
function F=fcastest(T,V,laplacienV);
load donnee2 b e1 p1 t1;
[gradtx,gradty]=pdegrad(p1,t1,T);%aux centres de gravité
[gradvx,gradvy]=pdegrad(p1,t1,V');
gradscalaire=pdeprtni(p1,t1,(gradtx.*gradvx+gradty.*gradvy));

F=(-sigmatest(T).*laplacienV'-gradscalaire.*derivsigmatest(T));
```

On finit par tout interpoler sur le maillage grossier dans le but d'une comparaison avec l'algorithme de Newton :

```
function [Tg,Vg,Fg]=solutionexacte(T,V,F);
load donnee2 p1 t1 p e t ;
Tg=griddata(p1(1,:),p1(2,:),T,p(1,:),p(2,:));
Vg=griddata(p1(1,:),p1(2,:),V,p(1,:),p(2,:));
Fg=griddata(p1(1,:),p1(2,:),F,p(1,:),p(2,:));
```

On reprend ensuite les fonctions propres à l'algorithme en changeant le κ et le σ, ce qui transforme alors également la fonction J,la fonction γ et les fonctions dérivées de κ et de σ. Finalement la fonction qui donnera la solution de l'algorithme de façon directe est :

```
function [Res,TSol,Vsol]=Resoldirecte(U,Imax)

load donnee2 b p e t C1 tr p1 t1 Bconnect
i=0;
close all;
[V,laplacienV]=Vcastest;
T=Tcastest;
F=fcastest(T,V,laplacienV);
[Tg,Vg,Fg]=solutionexacte(T,V,F);

Ff=pdeintrp(p,t,Fg');

tol=1e-8;
erreur=tol+1;

[v,v]=size(C1);
while(erreur > tol)&(i<Imax)
    S=-Ftest(U,Ff)'*Bconnect;

    on enlève les points extérieurs

    A1=Atest(U);
    E1=Etest(U,Ff);
    [B1,D1]=MatBD(U,Ff);
    h=(inv(C1)*(D1*inv(A1)*B1+E1)-speye(v))\S';
    h2=Bconnect*h;

    on remet les points extérieurs

    U=U+h2;
    i=i+1;
    erreur=norm(h)/norm(U)
end
TSol=U;
Res=norm(Tg'-TSol)/norm(TSol);
figure;
pdesurf(p,t,Tg');
figure;
pdesurf(p,t,TSol);
figure;
pdesurf(p,t,Tg'-TSol);
i
figure;
pdesurf(p,t,Vg');
sig=pdeintrp(p,t,sigmatest(TSol));
Vsol=assempde(b,p,e,t,sig,0,Ff);
figure;
pdesurf(p,t,Vsol);
```

retourne le nombre d'itérations et trace en fig. 4.2 la solution exacte, la solution renvoyée par l'algorithme, et la différence entre les deux solutions,en fig. 4.3 le potentiel V exact et le potentiel V_h trouvé grâce à l'algorithme

On prend comme point de départ de l'algorithme le vecteur nul aux noeuds du maillage grossier, l'algorithme converge alors en 7 itérations seulement, ce qui correspond à une dizaine de secondes. Et on obtient la solution de l'algorithme (qui retourne $U = J(T) = T$ aux noeuds du maillage grossier) tout à fait comparable à la solution exacte, avec un résidu relatif de 0.02. On trace aussi la solution V trouvée grâce au T de l'algorithme et `assempde` et la solution exacte.
Voici la comparaison graphique :

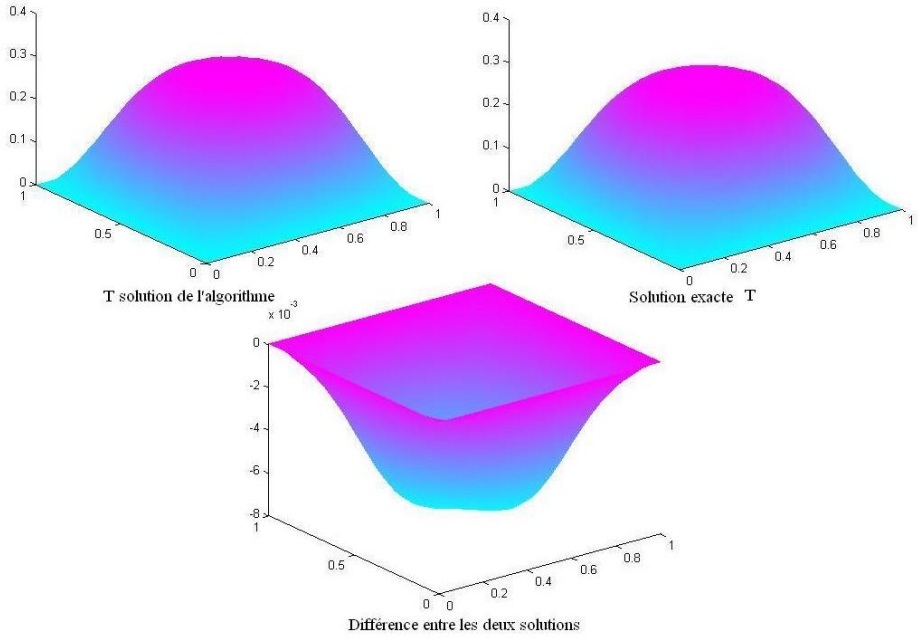


FIG. 4.2 – Comparaison entre le T exact et celui calculé par l'algorithme

Conclusion : les solutions exactes et celles que nous renvoie l'algorithme que nous avons programmé sont les mêmes à une différence de 10^{-3} près. De plus, nous avons essayé cet algorithme en prenant d'autres formules pour σ, et les résultats concordent tant que l'on peut calculer T par `pdenonlin`. La convergence est très rapide, et ce pour toutes les façons de résolution que nous avons envisagées (cf. **Chap. 5**). Ainsi, notre algorithme est juste et on va pouvoir l'appliquer au cas réel de notre projet.

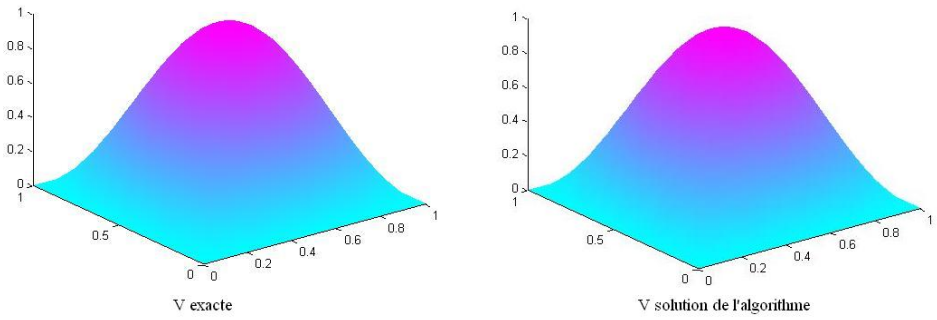


FIG. 4.3 – Comparaison entre le V exact et celui calculé par l'algorithme

Chapitre 5

Précisions sur les algorithmes utilisés

Au cours de notre projet, et de ce rapport, nous avons utilisé plusieurs notions mathématiques qui permettent entre autres de résoudre des problèmes numériquement. Tout d'abord, nous nous sommes servis plusieurs fois de la fonction `assempde` qui résout numériquement les E.D.P du type :

$$-\text{div}(a(x)\nabla x) + c(x)u(x) = f(x)$$

par la méthode des éléments finis.
Puis nous avons utilisé un algorithme d'optimisation, l'algorithme de Newton pour résoudre notre problème devenu un problème de point fixe : en notant $F(k) = \xi(k) - k$, nous voulions déterminer k tel que $F(k) = 0$. Nous expliquerons ici les mathématiques implicites à ces outils.

5.1 La méthode des éléments finis

Pour expliquer cette méthode qui est absolument cruciale car très utilisée dans le monde de la simulation numérique, nous prendrons l'exemple d'un problème concret : la diffusion à travers un mur. On considère alors le problème suivant :

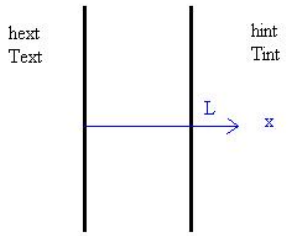


FIG. 5.1 – Schéma du système

$$\begin{cases} \rho c \frac{\partial T}{\partial t} = \lambda \frac{\partial^2 T}{\partial x^2} & \text{dans le mur} \\ T(x,0) = T_0 & \text{en } t = 0 \\ \lambda \frac{\partial T}{\partial x} = h_{ext}(T - T_{ext}) & \text{en } x = 0 (\text{condition de Neumann}) \\ \lambda \frac{\partial T}{\partial x} = -h_{int}(T - T_{int}) & \text{en } x = L (\text{condition de Neumann}) \end{cases}$$

Le problème peut se reformuler sous la forme intégrale, discrétisée tout d'abord en temps :

Trouver $T \in H^1[0,L]$tel que $\forall \Psi \in H^1[0,L]$

$$\int_t^{t+\Delta t} \int_0^L \Psi \rho c \frac{\partial T}{\partial t} dx dt = \int_t^{t+\Delta t} \int_0^L \Psi \lambda \frac{\partial^2 T}{\partial x^2} dx dt$$

On intervertit l'ordre des intégrations pour le membre de gauche et on intègre le second terme par parties :

$$\int_0^L \Psi \rho c (T^1 - T^0) dx = \int_t^{t+\Delta t} \left(\left[\Psi \lambda \frac{\partial T}{\partial x} \right]_0^L - \int_0^L \frac{\partial \Psi}{\partial x} \lambda \frac{\partial T}{\partial x} dx \right) dt$$

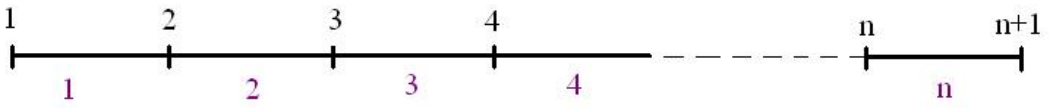
En notant $T^1 = T(t + \Delta t)$. Puis en supposant que pour Δt suffisamment petit, le gradient des fonctions Ψ et T est constant, égal à leur valeur en $t + \Delta t$ si l'on considère un schéma implicite, et en prenant en compte les conditions aux bords, on obtient une formulation faible de notre problème :

Trouver $T \in H^1[0,L]$tel que $\forall \Psi \in H^1[0,L]$

$$\int_0^L \Psi \rho c (T^1 - T^0) dx = - \int_0^L \Delta t \frac{\partial \Psi}{\partial x} \lambda \frac{\partial T^1}{\partial x} dx + \Delta t \left(\Psi(L) h_{int} (T^1 - T_{int}) - \Psi(0) h_{ext} (T^1 - T_{ext}) \right)$$

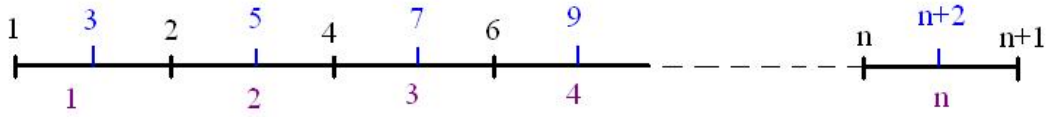
1. C'est ici qu'intervient la méthode des éléments finis : pour approximer les fonctions que nous devons intégrer pour résoudre ce problème.
   De façon générale, il y a trois étapes pour résoudre un problème à l'aide des éléments finis.

   – Il faut recouvrir notre domaine. En dimension deux, on choisit généralement des triangles dont la réunion recouvre tout le domaine si ce dernier est quarrable, sinon presque tout le domaine si l'on a affaire à un disque. En dimension une (c'est le cas ici) il suffit de prendre des segments pour recouvrir notre domaine qui est en fait une droite.
   On découpe alors notre domaine en segments que l'on numérote selon le schéma :



   Dans cette figure un segment a deux noeuds et la numérotation globale est logique. Cependant, il est possible d'attribuer plus de deux noeuds à un seul segment et de numéroter de façon moins "logique" :

   Dans notre exemple nous choisirons le cas le plus simple où un élément a seulement deux noeuds et tous les éléments ont la même longueur notée δx.



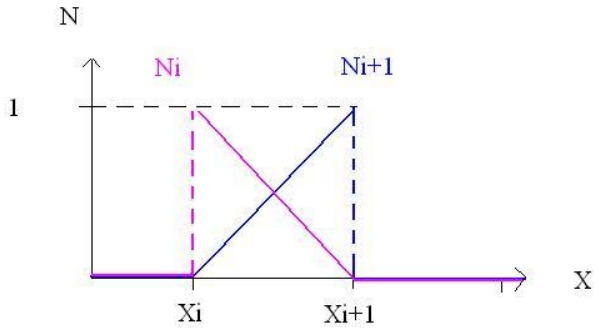
– Il faut ensuite définir ce qu'on appelle les fonctions "chapeaux", ou fonctions d'interpolation : Ces fonctions doivent respecter plusieurs critères afin de former une base de l'espace V_h, qui est l'espace des fonctions définies sur le recouvrement (qui n'est pas nécessairement inclus dans V).
Soit l'élément e, avec notre numérotation e est bordé à gauche par le noeud numéro e et à droite par le noeud numero $e+1$. Alors nous cherchons les fonctions chapeaux N_e et N_{e+1} telles que :

$$\begin{cases} N_i(x) = 1 & \text{si } x = x_i \\ N_i(x) = 0 & \text{si } x \neq x_i \\ N_i(x) = 0 & \text{si } x \notin e \\ \sum_{i \in e} N_i(x) = 1 & \forall x \end{cases}$$

Alors toutes nos fonctions seront définies par :

$$f(x) = f_e N_e(x) + f_{e+1} N_{e+1}(x) \text{ avec } f_e, f_{e+1} \text{ les valeurs aux noeuds e et e+1}$$

Il est alors possible de définir des fonctions polynomiales qui respectent ces critères, les plus faciles à mettre en oeuvre et les plus courantes sont les fonctions $\mathbb{P}_1$, c'est-à-dire des polynômes d'ordre 1.
Cela donne ici :



Alors on a ici :

$$\begin{cases} N_i(x) = \frac{x_{i+1} - x}{\delta x} & \text{sur } [x_i, x_{i+1}] \\ N_{i+1}(x) = \frac{x - x_i}{\delta x} & \text{sur } [x_i, x_{i+1}] \end{cases}$$

– Ensuite il faut mettre en place ces fonctions pour conduire finalement à un système linéaire matriciel. Nous allons étudier comment.

2. Pour revenir à notre exemple, la formulation intégrale précédente peut encore se dicrétiser en espace pour devenir :

$$\sum_e \int_e \Psi \rho c (T^1 - T^0) dx = -\Delta t \sum_e \int_e \frac{\partial \Psi}{\partial x} \lambda \frac{\partial T^1}{\partial x} dx - \Delta t \left(\Psi(L) h_{int} (T^1 - T_{int}) - \Psi(0) h_{ext} (T^1 - T_{ext}) \right)$$

Regardons de plus près un des termes de gauche à savoir $\frac{1}{\Delta t} \int_e \Psi \rho c T^1 dx$. Pour $x \in e$ on a

$$\Psi(x) = \Psi_e N_e(x) + \Psi_{e+1} N_{e+1}(x)$$

Et

$$T^1(x) = T^1_e N_e(x) + T^1_{e+1} N_{e+1}(x)$$

Le terme peut alors se mettre sous la forme :

$$\frac{1}{\Delta t} \int_e \Psi \rho c T^1 dx = \frac{1}{\Delta t} \int_e \rho c \left\langle \Psi_e, \Psi_{e+1} \right\rangle \begin{pmatrix} N_e(x) \\ N_{e+1}(x) \end{pmatrix} \left\langle N_e(x), N_{e+1}(x) \right\rangle \begin{pmatrix} T^1_e \\ T^1_{e+1} \end{pmatrix} dx$$

$$\frac{1}{\Delta t} \int_e \Psi \rho c T^1 dx = \frac{1}{\Delta t} \int_e \rho c \left\langle \Psi_e, \Psi_{e+1} \right\rangle \begin{pmatrix} N_e(x)^2 & N_e(x) N_{e+1}(x) \\ N_e(x) N_{e+1}(x) & N^2_{e+1}(x) \end{pmatrix} \begin{pmatrix} T^1_e \\ T^1_{e+1} \end{pmatrix} dx$$

Soit finalement

$$\frac{1}{\Delta t} \int_e \Psi \rho c T^1 dx = \left\langle \Psi_e, \Psi_{e+1} \right\rangle [M_e] \begin{pmatrix} T^1_e \\ T^1_{e+1} \end{pmatrix}$$

Avec

$$[M_e] = \frac{\rho c}{\Delta t} \begin{pmatrix} \int_e N_e(x)^2 dx & \int_e N_e(x) N_{e+1}(x) dx \\ \int_e N_e(x) N_{e+1}(x) dx & \int_e N^2_{e+1}(x) dx \end{pmatrix}$$

Ici

$$[M_e] = \frac{\rho c}{3 \Delta t \delta x} \begin{pmatrix} 1 & 1/2 \\ 1/2 & 1 \end{pmatrix}$$

Il faut ensuite sommer ces termes sur tous les éléments e, ce qui revient à assembler les matrices élémentaires $[M_e]$ pour donner la matrice de masse totale :

$$[M] = \frac{\rho c}{3 \Delta t \delta x} \begin{pmatrix} 1 & 1/2 & 0 & \cdots & 0 \\ 1/2 & 2 & 1/2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & 1/2 & 2 & 1/2 \\ 0 & \cdots & 0 & 1/2 & 1 \end{pmatrix}$$

On agit de même pour tous les termes de l'équation intégrale pour obtenir une équation de la forme :

$$\left\langle \Psi \right\rangle [A] (T) = \left\langle \Psi \right\rangle (B) \ \ \forall \Psi \in H^1[0, L]$$

D'où il faut finalement résoudre

$$[A] (T) = (B)$$

où $[A]$ est la somme d'une matrice masse, d'une matrice raideur et autres, en fonction des problèmes.

5.2 L'algorithme de Newton

Soit Ω un ouvert d'un espace vectoriel normé V et $J : \Omega \to \mathbb{R}$ une fonction donnée. On rappelle que si la fonction J possède un minimum relatif en un point $u \in \Omega$ et si elle est dérivable en ce point alors nécessairement $J'(u) = 0$. Dans ce paragraphe nous allons donc nous intéresser à la résolution de cette équation, c'est-à-dire en supposant la fonction J dérivable dans Ω, on cherchera les zéros de l'application dérivée $J' : \Omega \to V'$. On utilisera ici un algorithme d'approximation d'une telle solution a, c'est-à-dire la construction d'une suite (x_k) de point de Ω telle que $lim_{k\to\infty} x_k = a$.

La méthode de Newton consiste à construire cette suite pour minimiser la fonction f telle que

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \text{ pour } k \geq 0$$

avec $x_0 \in \Omega$ arbitraire.
Graphiquement, chaque point x_{k+1} est l'intersection de l'axe des abscisses avec la tangente à la courbe en x_k (en dimension une), fig. 1.2.

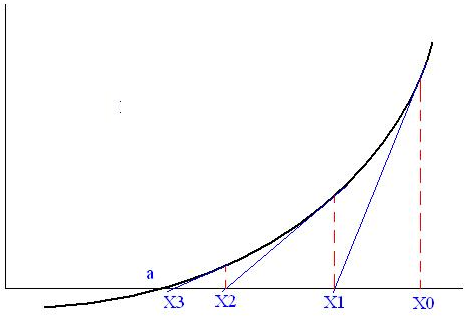


FIG. 5.2 – Schéma de principe

Ce cas particulier suggère la définition suivante de la méthode de Newton :

$$\begin{cases} f : \Omega \in X \to Y \\ x_0 \in \Omega \text{ donné} \\ x_{k+1} = x_k - f'(x_k)^{-1} f(x_k), \quad k \geq 0 \end{cases}$$

Ce qui suppose que tous les points x_k restent dans Ω et demande une vérification, que l'application f est dérivable dans Ω et que sa dérivée est une bijection de X sur Y en tout point $x \in \Omega$.

Dans le cas de systèmes d'équations non linéaires, la méthode de Newton s'applique aussi mais sous la forme :

$$\begin{cases} f'(x_k)\delta x_k = -f(x_k) \\ x_{k+1} = x_k + \delta x_k. \end{cases}$$

On conçoit que pratiquement, il soit très couteux à chaque itération de calculer les éléments de la matrice $(\partial_j f_i(x_k))$ du système précédent et de résoudre ensuite le système linéaire correspondant. Par ailleurs, si la méthode est convergente, les vecteurs x_k consécutifs doivent peu différer, de même que les matrices correspondantes. Ces considérations conduisent naturellement à une variante de la

méthode de Newton qui consiste à conserver la même matrice pendant p itérations consécutives :

$$\begin{cases} x_{k+1} = x_k - f'(x_0)^{-1} f(x_k), & 0 \leq k \leq p-1 \\ x_{k+1} = x_k - f'(x_p)^{-1} f(x_k), & p \leq k \leq 2p-1 \\ \vdots \end{cases}$$

On peut également remplacer p par ∞ ce qui conduit à des itérations du type $x_{k+1} = x_k - f'(x_0)^{-1} f(x_k)$ où $f'(x_0) = A_0$ une matrice inversible. On aura alors une convergence du type : (voir figure)

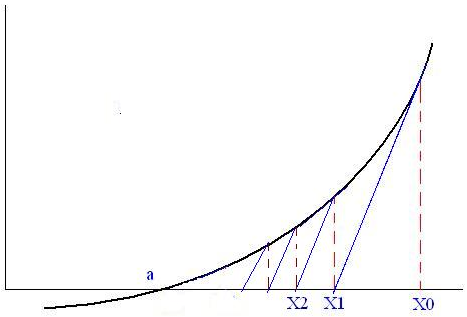


FIG. 5.3 – Avec matrice constante

Dans le cas de $A_0 = I$, la méthode s'appelle méthode des approximations successives, où l'on résout l'équation $f(x) = 0$ en posant $x = x - f(x)$. Cette méthode itérative particulière nous donne l'occasion de montrer les difficultées rencontrées dans ce genre de méthode (méthode de Newton incluse). Prenons par exemple la résolution de

$$f(x) = x^2 - \frac{1}{4} = 0, \text{ de racines } x = \frac{1}{2} \text{ et } x = -\frac{1}{2}$$

de sorte que la résolution par notre méthode donne :

$$f(x) = 0 \Leftrightarrow x = x - f(x) \Leftrightarrow x = -x^2 + x + \frac{1}{4}$$

On représente les itérations successives sur la figure, pour différentes valeurs de x_0.

On peut donc faire la liste des différentes possibilités :

$$x_0 < -\frac{1}{2} : \text{la méthode diverge}$$

$$x_0 \in \left\{-\frac{1}{2}\right\} \cup \left\{\frac{3}{2}\right\} : lim_{k\to\infty} x_k = -\frac{1}{2}$$

$$-\frac{1}{2} < x_0 < \frac{3}{2} : lim_{k\to\infty} x_k = \frac{1}{2}$$

$$x_0 > \frac{3}{2} : \text{la méthode diverge}$$

On voit donc apparaître sur cet exemple, d'une part le fait que la méthode converge seulement si la valeur initiale x_0 est suffisamment proche d'une racine, et d'autre part le fait que la méthode

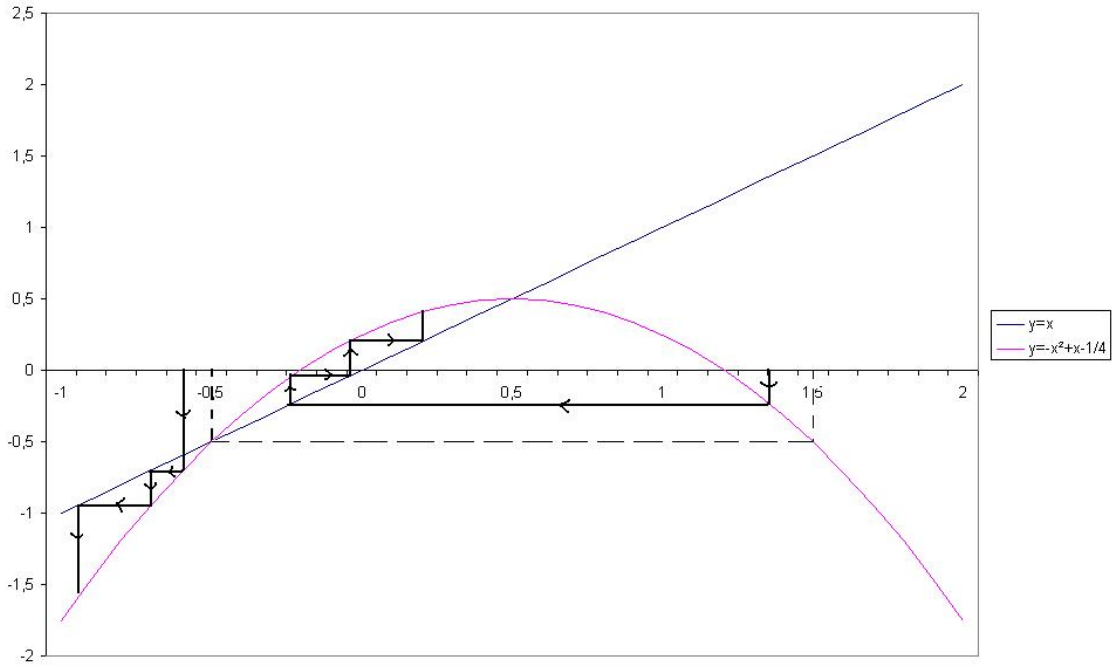


FIG. 5.4 – Influence du point de départ

converge presque tout le temps vers la même racine. En effet, seules deux valeurs de x_0 conduisent à la première racine, alors que tout un intervalle conduit à la seconde.
Le problème du choix de la valeur initiale est donc très important, il en dépend la convergence des algorithmes. Pour l'algorithme de Newton, il faut donc se placer suffisamment proche de la solution, solution qu'on ne connait pas à priori.

Par la suite, comme nous l'avons dit précédemment, la résolution à chaque étape de l'équation $f'(x_k)\delta x_k = -f(x_k)$ est très coûteuse, on prendra souvent une approximation de la matrice $(\partial_j f_i(x_k))$ notée $A_k(x)$, alors il existe un théorème qui, sous plusieurs hypothèses, nous montre que l'algorithme doit converger vers la solution.

Enoncé du théorème
On suppose l'espace X ***complet et la fonction*** $f : \Omega \in X \to Y$ ***dérivable dans l'ouvert*** Ω**.**
On suppose par ailleurs qu'il existe trois constantes r**,** M**,** β ***telles que***

$$r > 0, \ \textbf{et}\ B = \{x \in X; \|x_{x0}\| \leq r\} \subset \Omega$$

$$\underbrace{sup}_{k\geq 0}\ \underbrace{sup}_{x\in B} \left\|A_k^{-1}(x)\right\| \leq M$$

$$\underbrace{sup}_{k\geq 0}\ \underbrace{sup}_{x,x'\in B} \left\|f'(x) - A_k^{-1}(x')\right\| \leq \frac{\beta}{M}\ \textbf{et}\ \beta < 1$$

$$\|f(x_0)\| \leq \frac{r}{M}(1-\beta).$$

Alors la suite $(x_k)_{k\geq 0}$ ***définie par***

$$x_{k+1} = x_k - A'_k(x_{k'})^{-1} f(x_k), \ k' \geq 0$$

est entièrement contenue dans la boule B et converge vers un zéro de f, qui est le seul zéro de f dans la boule B. Enfin, la convergence est géométrique :

$$\|x_k - a\| \leq \frac{\|x_1 - x_0\|}{1-\beta}\beta^k.$$

Il existe d'autres théorèmes similaires sur la convergence de l'algorithme de Newton, dont les hypothèses sont également restrictives. Il faut néanmoins tenir compte de la fonction f qui peut avoir des propriétés facilitantes comme la convexité ou l'ellepticité par exemple. Dans le cas général, il faut retenir que le choix du point de départ de l'algorithme est crucial.

5.3 Les différentes résolutions de l'algorithme de Newton

Au coeur de l'algorithme de Newton, il faut résoudre à chaque itérations :

$$\begin{cases} f'(x_k)\delta x_k = -f(x_k) \\ x_{k+1} = x_k + \delta x_k. \end{cases}$$

Et trouver le δ nécessite l'inversion lourde de matrices, ce qui n'est pas judicieux pour le temps de calcul. Pour remédier à ce problème, on a envisagé, en programmant le cas test, différentes résolutions possibles.
Dans notre problème, cette étape consiste à trouver h tel que :

$$C^{-1}(DA^{-1}Bh + Eh) - h = S$$

Il faut donc inverser tout d'abord A, puis C et enfin le produit matriciel $C^{-1}(DA^{-1}B + E) - Id$, ce qui se révèle très long, et nécessite une grande mémoire si l'on a un maillage fin.
Cette méthode directe a déjà été présentée dans le cas test, où on a eu la chance d'avoir une convergence rapide.

Nous avons ensuite programmé trois autres méthodes pour résoudre ce problème, qui se différencient par le nombre d'itérations, par le fait que la convergence de certaine n'est pas toujours sûre, ce qui nous laisse le choix d'utiliser celle qui marche le mieux en fonction du cas étudié.

– **La résolution indirecte**
Comme on souhaite trouvé h tel que $C^{-1}(DA^{-1}Bh + Eh) - h = S$, on va imaginer un algorithme itératif qui calculera cela en cherchant

$$h_{n+1} = C^{-1}(DA^{-1}B + E)h_n - S$$

en posant $h_0 = 0$, et on continue les itérations tant que la différence entre deux h_n successives est supérieure à une certaine tolérance. Voici le programme à insérer dans la fonction finale de résolution, qui permet par cette méthode de n'inverser aucune matrice :

```
n=0;
h=zeros(v,1);
delta=1;
while(norm(delta)>10^(-15))&&(n<1000)
```

on pose donc

```
        delta=h;
```

puis on résout $dn + 1 = C^{-1}(DA^{-1}B + E)dn - S$

on pose z *tq* $A1z = B1h$

```
        z=A1\(B1*h);
```

on pose w *tq* $C1w = D1z + E1h$ *car alors* $W = C1^{-1}(D1A^{-1}B1h + E1h)$

```
        w=C1\(D1*z+E1*h);
        h=w-S';
        delta=h-delta;
        n=n+1;
    end
```

Dans le cas test, cette méthode converge aussi rapidement que la méthode directe en nombre d'itérations, mais le temps de calcul est plus court car les matrices de taille environ 400×400 ne sont jamais inversées directement. On obtient évidemment les mêmes résultats en ce qui concerne les solutions, cependant, comme cette résolution est de la forme d'une suite géo-arithmétique ($U_n = aU_{n-1} + b$) où $a = C^{-1}(DA^{-1}B + E)$, il y a des conditions sur la matrice a pour que la méthode converge. En effet, si le $||a||_2 > 1$, la méthode a de grandes chances de ne pas s'approcher de la solution cherchée.

– **méthode GMRES**

Une autre approche pour calculer nos inverses de matrices consiste à utiliser une fonction intermédiaire, la fonction `GMRES` de Matlab qui calcule elle-même l'inverse de $C^{-1}(DA^{-1}B + E) - Id$ par une méthode itérative. On crée tout d'abord la fonction *prodmat* qui va calculer ce produit matriciel, et on appellera alors alors cette fonction dans la fonction principale de l'algorithme.
Voici le fonction `prodmat` :

```
function y=prodmat(x,R,D,A,B,E)
```

Nom : prod
Description : réalise le produit matrice vecteur du premier membre du système linéaire à résoudre à chaque étape, de manière moins coûteuse et pour pouvoir appliquer la méthode GMRES
Input : x vecteur aux degrés de liberté, R factorisée de Choleski de la matrice C, D, A, B, E les matrices construites à chaque étape
Output : produit $y = (inv(C) * (D * inv(A) * B + E) - I_n)x$

```
aux=D*inv(A)*B+E;
y=R\(R'\(aux*x))-x;
```

Puis on remplace la ligne de commande de la fonction *Resoldirecte* correspondant au calcul de h par :

```
[h,flag,res]=gmres('prodmat',S',100,1e-8,10,[],[],zeros(v,1),R,D1,A1,B1,E1);
```

On obtient les mêmes résultats en un même nombre d'itérations. Cependant, dans notre cas test où la taille des matrices n'était que de 400×400, le gain de temps n'est pas visible. En effet, GMRES devient une méthode intéressante si le système matriciel est beaucoup plus important (de l'ordre de 10000×10000), elle est donc à envisager si l'on souhaite affiner le maillage.

– **Newton à pas optimal**

On peut aussi régler le problème du temps de calcul en modifiant la méthode directe qui est la seule méthode qui converge toujours. En effet, la méthode originale de Newton consiste à faire

$$\begin{cases} f'(x_k)\delta x_k = -f(x_k) \\ x_{k+1} = x_k + \delta x_k. \end{cases}$$

Où l'on choisit a priori arbitrairement d'ajouter δ à chaque itération. Rien n'empêcherait d'ajouter plutôt $\rho\delta$ où $\rho \in [0,1]$, ce qui permettrait de converger plus rapidement car la différence entre deux solutions serait plus petite, une fois que l'on s'approche de la solution cherchée. Il reste alors à choisir le meilleur coefficient multiplicateur ρ. Si nous revenons à notre problème initial, c'est à dire que l'on cherche le vecteur U tel que $\xi(U) - U = 0$, alors le meilleur ρ sera celui qui permet de minimiser $\xi(U + \rho h) - (U + \rho h)$.
On a donc programmé une fonction qui dépend de ce coefficient ρ et que l'on minimisera dans notre algorithme de résolution final :

```
function fct_opt=optimal(s,U,Ff,h,Bconnect)
Nom : optimal
Input : le s minimiseur, le vecteur solution U aux noeuds, la fonction Ff aux centres de gravité des
triangles, le vecteur h cherché, et la matrice Bconnect.
Output : la fonction que l'on souhaite minimiser.

F1=Ftest(U+s*h,Ff)'*Bconnect;
fct_opt=norm(F1);
```

Puis dans notre algorithme de résolution, il suffit de définir ρ comme le minimiseur de la fonction *optimal* et de remplacer la commande corresponadant à $U_{k+1} = U_k + \delta_k$.
On donne ici la boucle principale :

```
while(erreur > tol)&(i<Imax)
    S=-Ftest(U,Ff)'*Bconnect;
    A1=Atest(U);
    E1=Etest(U,Ff);
    [B1,D1]=MatBD(U,Ff);
    h=(inv(C1)*(D1*inv(A1)*B1+E1)-speye(v))\S';
    h2=Bconnect*h;
    rho=fminbnd(@optimal,0.1,1,[],U,Ff,h2);
    calcule le minimiseur de la fonction optimal entre 0 et 1
    U=U+rho*h2;
    i=i+1;
    erreur=norm(h)/norm(U)
end
```

Cette résolution se fait en 6 itérations pour le cas test, en partant du vecteur nul, donc on gagne une itération avec cette méthode. On remarque que dans ce cas test, cette méthode n'est pas vraiment utile puisque les coefficients ρ calculés à chaque itérations sont très proches de 1. Cependant, elle peut s'avérer très utile dans d'autres cas, c'est pourquoi il est préférable d'avoir plusieurs méthodes à sa disposition.

Conclusion

Ce projet a été pour nous l'occasion de découvrir le monde du calcul scientifique. De la modélisation physique au traitement des résultats numériques, ce projet nous a permis de comprendre et de saisir toute la complexité tant théorique que pratique que ce type de problème revêt. La résolution d'équations aux dérivées partielles nécessite à la fois un bagage mathématique théorique relativement conséquent mais aussi de bonnes connaissances algorithmiques et informatiques. Nous avons bien sûr rencontré de nombreuses difficultés, en particulier au niveau de l'implémentation informatique de la méthode et du remplissage des matrices. Une compréhension plus profonde de la méthode a été nécessaire pour résoudre ces difficultés. De même, un tel projet nécessite, de par sa taille, une architecture et une organisation informatique que nous n'avions pas l'habitude de rencontrer. Il nous a donc fallu apprendre à évoluer dans une telle complexité et, plus d'une fois, il nous a fallu se plonger en profondeur dans les algorithmes et dans le fonctionnement de Matlab pour résoudre les problèmes rencontrés. Les enseignements de ce projet ont donc été nombreux et riches. Il nous a permis de faire le lien entre la théorie des E.D.P que nous avions abordée au premier semestre et la résolution effective de ces dernières. Deux domaines qui, bien que complémentaires, n'exigent pas les mêmes méthodes de travail et d'approche. D'un point de vue informatique, ce projet nous a permis d'acquérir une connaissance assez globale du logiciel Matlab et d'approfondir notre pratique de la méthodes des éléments finis qui nous avons découverte tout au long du second semestre. Enfin, d'un point de vue purement technique, la rédaction de rapport a aussi été l'occasion de découvrir le formidable outil de traitement de texte qu'est LaTeX.

Bien sûr, le monde de la résolution des E.D.P étant tellement vaste et complexe, nous n'avons qu'effleuré ce dernier et bien d'autres aspects de ce projet auraient pu se révéler intéressants à développer. Par exemple, nous avons choisi de nous restreindre plus à la résolution effective numérique du système plutôt que de se pencher sur la théorie et justifier l'existence de solutions faibles pour ce problème. De même, il existe d'autres méthodes de résolution de tels systèmes (méthode du point fixe par exemple), leur implémentation aurait pu se révéler fructueuse. Enfin, l'analyse numérique de nos algorithmes pourrait elle aussi être judicieuse et améliorer l'efficacité et la convergence de ces derniers. Malheureusement, faute de temps et aussi face aux difficultés rencontrées, nous n'avons pas eu l'opportunité de développer en profondeur ces aspects. Néanmoins, tout l'intérêt de ce projet réside justement dans l'ouverture vers ces aspects. Par le travail effectué, par les lectures, nous avons abordé un monde vaste et passionnant qui a éveillé tout notre intérêt.

Bibliographie

[1] Adrian Biran et Moshe Breiner, *MATLAB pour l'ingénieur versions 6 et 7*, Pearson Education, 2004
Introduction au logiciel Matlab. De niveau élémentaire, les très nombreux exemples et codes donnés permettent de bien prendre en main les versions 6 et 7 du logiciel. N'inclut pas les modules de Matlab.

[2] Michel Goossens, Frank Mittelbach et Sebastian Rahtz, *The LaTeX GRAPHICS COMPANION : Illustrating Documents with TeXand PostScript*, Addison Wesley, Reading, 1997
Complément graphique de [3]. D'un niveau assez avancé, il est recommandé de bien connaître [3] avant de se lancer dans les graphiques en TeXde ce livre.

[3] Michel Goossens, Frank Mittelbach, Alexander Samarin, *LaTeX COMPANION*, CampusPress, 2001
Bible de LaTeX, ce livre s'adresse à la fois aux débutants et aux confirmés de LaTeX. Très complet, tous les codes sources sont donnés et les explications sont très claires. Livre indispensable pour quiconque veut écrire en LaTeX.

[4] François Rouvière, *Petit guide de calcul différentiel à l'usage de la licence et de l'agrégation*, Vuibert, Cassini, 2003
Rappels de cours et nombreux exercices de calcul différentiel. Nous a permis de clarifier la notion de différentielle et les différentes notions liées au calcul différentiel et à l'algorithme de Newton.

[5] Philippe G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimisation*, Dunod, Sciences Sup, 2002
Ce livre présente la plupart des algorithmes classiques que l'on rencontre en calcul scientifique. Très complet tant du point de vue théorique (nombreux complément d'analyse numérique matricielle, algorithme de Newton...) que pratique. Les exemples sont parfaitement clairs.

[6] Antoine Henrot, *Outils Numériques I*, Ecole des Mines de Nancy, 2006
Notes de cours sur les éléments finis et les différences finies. Mise en oeuvre des méthodes. Clair et concis

[7] Graham Wohan, *The Cambridge Handbook of Physics Formulas*, Cambridge University Press, 2000
Formulaire de sciences physiques avec de nombreux rappels de mathématiques. Très pratique et facile d'accès

Table des figures