Utilisation de la Méthode des Solutions Manufacturées pour la vérification logicielle

Résumé :

Ce document présente l'utilisation de la Méthode des Solutions Manufacturées pour la validation logicielle de *Code_Aster*. Le principe de cette méthode est détaillé sur un exemple simple de diffusion thermique. Cet exemple nous permet de préciser l'intérêt de cette approche et particulièrement de présenter la démarche de validation qui peut être mise en œuvre. Enfin, l'implantation informatique de la Méthode des Solutions Manufacturées, basée sur l'utilisation du calcul symbolique, est détaillée.

Table des Matières

1 Introduction

La vérification d'un logiciel scientifique consiste à s'assurer de l'absence d'erreur dans la programmation de la modélisation implantée. Une approche classique et très fiable consiste à comparer le résultat du logiciel à celui d'une solution analytique. Malheureusement, pour les problèmes complexes et/ou non-linéaires, les solutions analytiques deviennent très difficile à obtenir. C'est dans ce cadre que la Méthode des Solutions Manufacturées est particulièrement intéressante.

La Méthode des Solutions Manufacturées est une méthode de vérification de logiciels scientifiques. Il s'agit d'une méthode systématique d'obtention de solutions analytiques pour des problèmes qui peuvent être très complexes et non-linéaire. Son principe est simple : on se donne explicitement la solution à chercher sous la forme d'une expression analytique et, à partir de cette solution, on construit les données (blocages, chargements) nécessaires à l'obtention de cette solution.

Il est intéressant de noter que cette méthode est très couramment utilisée en mécanique des fluides. Ainsi l'«American Institute of Aeronautics and Astronautics» (AIAA) l'a incluse dans ses normes d'assurance qualité logicielle en 1998 [bib1] Cette pratique est beaucoup plus rare en mécanique des solides ; ainsi l'«American Society of Mechanical Engineers» (ASME) ne l'a intégrée dans sa norme qu'en 2006 [bib2], soit beaucoup plus récemment. Cela n'est pas lié à une difficulté particulière, des études poussées en mécanique des solides ayant été menées grâce à cette méthode [bib3].

Dans la suite de ce document, nous allons exposer le principe de cette méthode sur un exemple simple puis la démarche de validation applicable. Nous détaillerons ensuite la mise en œuvre informatique dans *Code_Aster*.

2 Principe général et utilisation pour la vérification

2.1 La Méthode des Solutions Manufacturées en thermique

Le principe de cette méthode est très simple ; nous allons l'illustrer sur un exemple de thermique linéaire 2D. Avant tout, nous rappelons les équations d'un problème de thermique linéaire. Dans le domaine Ω , on recherche le champ de température $T(x,y)$ tel que :

$$\begin{cases} -\lambda \Delta T = s \text{ dans } \Omega \\ T = T_d \text{ sur } \partial\Omega_d \\ \lambda \vec{\nabla} T \cdot \vec{n} = q_n \text{ sur } \partial\Omega_n \end{cases} \quad (1)$$

où T_d et q_n sont respectivement la température et le flux imposés sur des parties de la frontière.

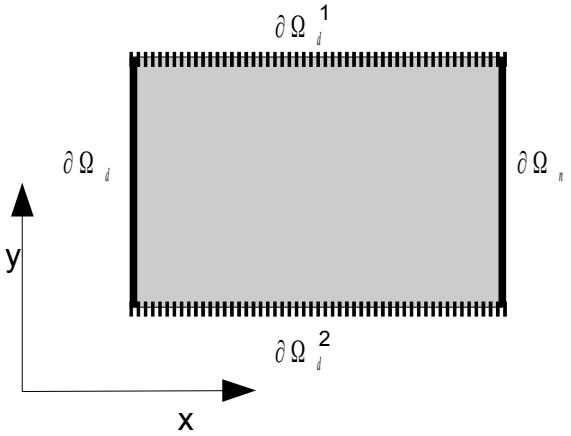
Les étapes de la Méthode des Solutions Manufacturées sont les suivantes :

- On choisit un domaine d'étude 2D arbitraire Ω comportant les frontières $\partial\Omega_d$ avec conditions de Dirichlet et $\partial\Omega_n$ avec conditions de Neumann. Par choisir, on entend « choisir la forme du domaine ». Ainsi nous choisissons la forme de la figure 1.
- On choisit une solution arbitraire, disons $T(x,y) = x^3 + y^3$
- En ré-injectant l'expression précédente dans (1), on déduit les expressions :
  - du terme source ;
  $$s(x,y) = -\lambda \Delta T = -\lambda (6x + 6y) \quad (2)$$
  - du terme de Dirichlet ;
  $$T_d(x,y) = x^3 + y^3 \quad (3)$$
  - du terme de Neumann :
  $$q_d(x,y) = \lambda \vec{\nabla} T(x,y) \cdot \vec{n} \quad (4)$$

Si on veut imposer cette condition sur le bord $\partial\Omega_n$, alors on on connait l'expression de la normale $\vec{n}$ à savoir $\vec{n}=[1,0]^T$. On obtient donc :

$$q_d(x,y)=\lambda\vec{\nabla}T(x,y)\cdot\vec{n}=3\lambda x^2 \quad (5)$$

Arrivé à ce stade, on dispose des informations nécessaires à la modélisation du problème étudié. Il est néanmoins essentiel de faire la remarque suivante : pour obtenir la solution recherchée, il faut bien affecter des conditions aux limites sur tout le bord $\partial\Omega$ du domaine d'étude. Au choix, suite à une partition de la frontière, cela peut être un mélange de conditions de Dirichlet et de Neumann. L'obligation d'imposer des conditions sur toute la frontière est notamment dû au fait qu'une absence de condition aux limites est implicitement une condition de Neumann nul. Or l'équation 3 montre que l'on n'a pas le choix de la valeur de cette condition. Dans l'exemple présent, nous faisons le choix d'imposer une condition de Neumann sur $\partial\Omega_n$ et des conditions de Dirichlet sur $\partial\Omega_d$, $\partial\Omega_d^1$ et $\partial\Omega_d^2$.



Dessin 1: Domaine d'étude

Une fois, les équations précédentes obtenues, on passe à la phase modélisation. On commence par choisir une discrétisation éléments finis V_h (linéaire, quadratique). On modélise dans son logiciel d'éléments finis préféré le domaine Ω_h auquel on applique les chargements $s_h(x,y)$, $T_{dh}(x,y)$ et $q_{dh}(x,y)$. Après résolution, on obtient la solution approchée $T_h(x,y)$.

2.2 Techniques de vérification

Grâce à la MSM, différentes vérifications sont réalisables.

2.2.1 Choisir une solution de référence dans l'espace d'approximation $T\in V_h$

C'est la vérification la plus simple car on doit alors obtenir $T(x,y)=T_h(x,y)$. A noter que, du fait de résolution numérique, on n'aura pas égalité parfaite mais la différence doit être de l'ordre de la précision machine $\sim 1.E-15$. De manière concrète, si on utilise des éléments linéaires, il faudra alors choisir une solution linéaire.

2.2.2 Choisir une solution de référence hors de l'espace d'approximation $T\notin V_h$

Dans le cas de figure où $T\notin V_h$, on n'a plus l'égalité du paragraphe précédent. Par contre, on peut assez facilement mesurer la vitesse de convergence de T_h vers T quand $h\to 0$.

On précise que la vitesse de convergence, avec la finesse h du maillage, de la solution calculée vers la solution analytique, dans une norme donnée, se définit comme le plus grand réel $\alpha>0$ tel que $\|T(x,y)-T_h(x,y)\|<C*h^{\alpha}$ où C est indépendant de h.
Pour mesurer la vitesse de convergence, on procède comme suit :

- On réalise des maillages de plus en plus fins du domaine Ω
- Pour chacun des maillages, on détermine la solution T_h
- Pour chacun des maillages, on mesure dans une norme bien choisie l'erreur $e_h = \|T(x,y) - T_h(x,y)\|$
- On calcule l'ordre de convergence asymptotique de e_h vers 0 quand $h \to 0$ par la formule $-\log(\frac{e_{h/2}}{e_h})/\log(2)$
- On vérifie de l'ordre de convergence est en accord avec la théorie.

Par exemple, pour une solution T suffisamment régulière, en utilisant le norme L_2, l'ordre de convergence d'un élément linéaire est au minimum de 2 tandis l'ordre de convergence d'un élément quadratique est au minimum de 3.

2.3 Aspects informatiques

2.3.1 Calcul symbolique

A la lecture des grands principe de la méthode, il est évident que le passage de l'expression analytique de la solution recherchée aux données d'entrées du problème éléments finis à résoudre nécessite de nombreuses opérations de dérivation et d'algèbre. Pour faciliter et automatiser cette étape, on s'appuie des fonctionnalités de calcul symbolique. Elles sont fournies par le module Sympy [bib4] du langage Python. Il s'agit d'un module intégralement écrit en Python d'une grande simplicité d'utilisation.

Pour faciliter la mise en œuvre de la MSM a été développée une classe Python TensorModule qui définit l'objet tenseur ainsi que ses méthodes (produit simplement et doublement contracté, trace, déterminant, …). Nous avons aussi développé les principaux opérateurs s'appliquant sur ces tenseurs (gradient, gradient symétrique, divergence, Laplacien). Sur la base de cette classe, nous avons défini un autre module HookTensor qui permet simplement d'implanter différents tenseurs de Hook (isotrope, anisotrope, orthotrope avec angles nautiques).

Ces classes sont placés dans le répertoire `bibpyt/Utilitai` des sources de Code_Aster.

2.3.2 Mise en œuvre dans Code_Aster

Dans cette partie, nous décrivons comme la MSM est mise en œuvre dans le langage de commandes de Code_Aster. Pour ce faire, nous nous basons sur l'exemple de thermique linéaire précédent et détaillons les principales étapes nécessaires. La mise en oeuvre précise est réalisée dans le cas-test `tplp107a`.

```
import sympy
```

On importe le module de calcul symbolique Sympy, pour utiliser ses fonctionnalités.

```
X,Y=sympy.symbols('XY');
```

On définit les variables d'espace qui seront utilisées pour le calcul symbolique.

```
T=sympy.Function('T');
S=sympy.Function('S');
```

On définit le nom des fonctions ; T sera bien évidemment la température et f le terme source.

```
T=100*(X**6+Y**6);
```

On donne l'expression de la solution manufacturée que l'on cherche à retrouver.

```
S=-Lambda*(sympy.diff(sympy.diff(T,X),X)+sympy.diff(sympy.diff(T,Y),Y));
```

On déduit l'expression du terme source de l'équation (2).

```
SS=FORMULE(NOM_PARA=('X','Y'),VALE=str(S));
```

Cette commande permet de transformer l'expression précédente en objet formule de Code_Aster qui pourra être évaluée dans le logiciel de la manière suivante :

```
CLIMIT=AFFE_CHAR_THER_F(MODELE=MO,
                        SOURCE=_F(GROUP_MA='SURFACE',
                                  SOUR=SS,),
                        );
```

3 Bibliographie

1 American Institute for Aeronautics and Astronautics, *Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*, 1998.

2 American Society of Mechanics Engineers, *Guide for Verification and Validation in Computational Solid Mechanics*, 2006.

3 Eric Chamberland, André Fortin, Michel Fortin, *Comparison of the performance of some finite element discretizations for large deformation elasticity problems*, Computers & Structures, Vol. 88, no. 11-12, pp. 664-673, 2010

4 Module Sympy de Python, http://code.google.com/p/sympy/