Version 1.5.2
conseils aux développeurs

13 février 2007

La version 1.5.2 est un premier pas vers l'application des spécifications développées dans la NT LMDL 2006-010 à Trio_U.

Les principaux changements apportés à la version sont les suivants :

On essaie de respecter au maximum la hiérarchie coupleur / problème.

Le coupleur peut accéder aux problèmes qu'il couple, mais les problèmes n'ont pas accès au coupleur ni aux autres problèmes (chacun ses oignons !).

Dans cet esprit, on a supprimé la référence que portait un problème vers son éventuel coupleur. Merci de ne pas la réintroduire...

Il reste des références croisées d'un problème à l'autre, en particulier pour les couplages en température (`Champ_front_contact_VEF`, `Echange_contact_VDF`). On a cantonné leur utilisation à une seule méthode (`updateGivenFields`) Le travail va maintenant porter sur la suppression de ces raccourcis : le coupleur aura la charge de récupérer les infos d'un problème et de les transmettre à l'autre. Appuyez-vous le moins possible sur les références que les problèmes portent les uns sur les autres, et évitez surtout de vous en servir hors de `updateGivenFields`.

Toujours dans le but de séparer les données relatives à chaque problème, et comme les valeurs de t, dt, facsec, ... sont stockées dans le schéma en temps et peuvent différer d'un problème à l'autre dans le cas de problèmes couplés, on associe un schéma en temps à chaque problème. En pratique, quand on associe un schéma au Probleme_Couple, il est cloné pour chacun des problèmes.

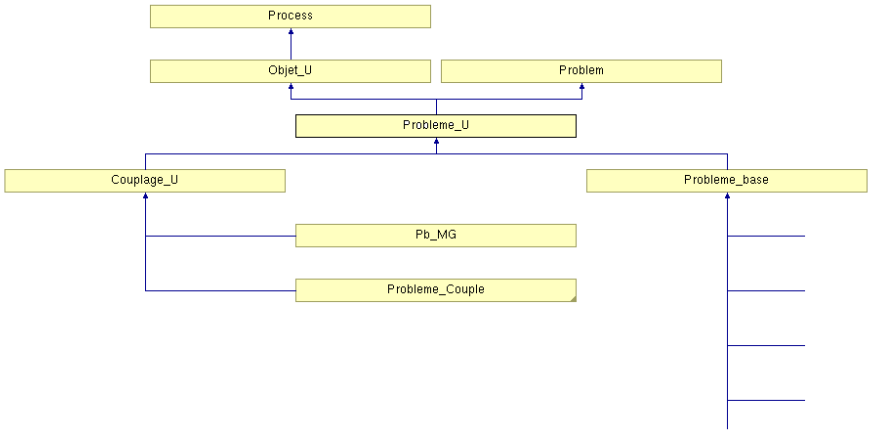
Une nouvelle fonctionnalité a été ajoutée à Probleme_Couple : il est possible de définir si on veut que des problèmes avancent ensemble (c'est-à-dire échangent des valeurs, puis font chacun une itération) ou les uns après les autres (pb1 fait une itération, puis pb2 a accès aux nouvelles valeurs pour mettre à jour ses champs d'entrée, puis pb2 fait une itération,...). Cela se fait à la lecture du Probleme_Couple en définissant des groupes de problèmes. Les groupes avancent l'un après l'autre, mais tous les problèmes à l'intérieur d'un groupe avancent ensemble.

La hiérarchie des problèmes est refondue pour faire apparaître l'API de couplage.

Les méthodes de l'API de couplage décrite dans la note LMDL 2006-010 et explicitées ci-dessous sont regroupées dans la classe `Problem`. elle-ci pourrait être commune à de nombreux codes.

Quelques spécificités de Trio_U sont ajoutées dans la classe `Probleme_U` qui herite de `Problem` et de `Objet_U`. En particulier `updateGivenFields`().

Il y a ensuite deux sortes de `Probleme_U` : ceux qui représentent un unique problème héritent de `Probleme_base` et ceux qui couplent plusieurs problèmes héritent de `Couplage_U` (c'est là qu'on retrouve le bon vieux `Probleme_Couple`, mais aussi `Pb_MG`). Intérêt : maintenant les problèmes simples et les problèmes couplés disposent de la même interface, et du même `run`().



On essaie de piloter les problèmes via des méthodes standardisées (API).

Certaines méthodes sont `const`. Généralement il y a une bonne raison. Evitez de contourner l'obstacle quand le compilateur râle...

Voyons dans l'ordre comment se déroule la résolution d'un problème.

– `initialize()`

C'est là que sont regroupées toutes les initialisations en début de calcul.

Appelle notamment les anciens `preparer_calcul`.

Evitez de cacher des initialisations ailleurs, par exemple dans un `if(init)` ou `if(nb_pas_dt==0)` à l'intérieur d'une méthode appelée à chaque pas de temps. On en a supprimé beaucoup, il en reste encore...

Cette méthode ne doit pas accéder à des données externes au problème, sinon l'effet des initialisations dépendrait de l'ordre dans lequel elles sont effectuées.

– `computeTimeStep(bool& stop) const`

Renvoie deux informations. La valeur de retour est la valeur du prochain pas de temps souhaité. La valeur assignée au booléen en argument vaut `true` si le problème souhaite s'arrêter.

Actuellement, les schémas en temps sont écrits de telle façon que cette valeur est déjà calculée et stockée dans `dt_`. Elle est calculée lors des `initialize`, `validateTimeStep` et `abortTimeStep`. Elle pourrait aussi bien être calculée lors de `computeTimeStep` mais non stockée (la méthode est `const` pour assurer que deux appels successifs rendent le même résultat !).

– `initTimeStep(double dt)`

Cette méthode prépare le calcul sur le prochain pas de temps, c'est-à-dire sur]`t`,`t+dt`] où `t` est le temps courant auquel on peut accéder par `presentTime`(). Attention la valeur du pas de temps est imposée par l'argument, elle écrase la valeur stockée `dt_`. Dans les cas courants sans couplage, les deux ont la même valeur.

`initTimeStep` a pour rôle d'allouer et initialiser les champs futurs avant la résolution. Elle remplace les anciens `preparer_pas_temps` et initialise les valeurs de l'inconnue et des conditions aux limites futures aux mêmes valeurs que le présent. Elle effectue diverses "mises à jour" qui seront utiles lors du calcul. Pour les champs qui n'ont qu'une valeur temporelle, elle est censée être sur]`t`,`t+dt`].

– `updateGivenFields()`

Cette méthode met à jour les champs d'entrée du problème (conditions aux limites, termes sources,...) pour l'intervalle de résolution.

C'est là que sont regroupées toutes les utilisations de références d'un problème à un autre qui subsistent encore.

A terme, les champs d'entrée devraient soit être indépendants de l'extérieur (et donc mis à jour dans `initTimeStep`), soit être imposés par le coupleur. Cette méthode devrait donc disparaître.

– `solveTimeStep()`

Cette méthode est chargée d'effectuer la résolution sur le pas de temps.

En fait la résolution est faite en plusieurs itérations, jusqu'à convergence, via la méthode suivante :

– `iterateTimeStep(bool& converged)`

Effectue une itération de la résolution. Renvoie dans converged `true` si on a atteint la convergence pour le pas de temps courant (c'est toujours le cas en explicite), et en valeur de retour `true` si on peut continuer, `false` s'il faut abandonner le calcul sur cet intervalle (divergence).

Cette méthode calcule les valeurs futures de l'inconnue et de ce qui en dépend. Interdit de modifier autre chose ! (le présent, le passé, un autre problème...)

C'est ici qu'on appelle `derivee_en_temps_inco`, ...

– `validateTimeStep()`

Quand le calcul sur le pas de temps est fini, cette méthode est appelée pour avancer le temps présent (tourner les roues).

Elle n'est pas supposée modifier les valeurs des champs.

– `isStationary() const`

Lorsqu'un pas de temps a été validé, cette méthode indique si le problème a évolué ou non sur l'intervalle considéré.

– `abortTimeStep()`

Cette méthode est appelée à la place de `validateTimeStep` si le calcul est abandonné pour ce pas de temps (par exemple il a divergé).

C'est l'occasion de remettre à zéro des valeurs futures qui peuvent avoir divergé (l'inconnue, la pression,...).

Par contre le présent ne peut toujours pas être modifié (pas plus qu'ailleurs).

– `terminate()`

Cette méthode est appelée à la fin de l'exécution.

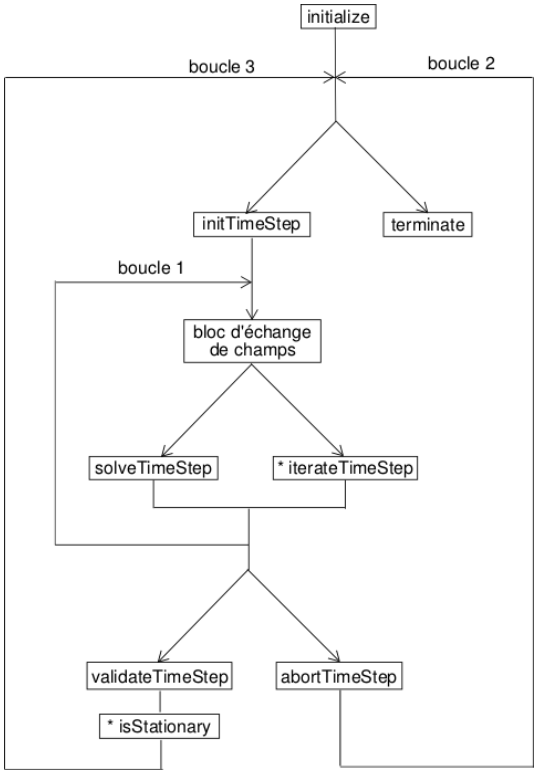


FIG. 1 – Diagramme d'exécution d'un problème dépendant du temps. Le signe étoile signifie : "un nombre quelconque d'appels à", ce nombre pouvant être nul. Le bloc d'échange de champs est remplacé pour le moment dans Trio_U par `updateGivenFields`.

On a gardé un seul `Resoudre`.

Il n'utilise que les méthodes standardisées des problèmes. Il est codé dans `Probleme_U::run()`
Il reste un `Resoudre_FT` qui devrait disparaître avec la nouvelle version du Front_Tracking.

On essaie de standardiser la description des schémas en temps.

Auparavant les schemas en temps n'étaient caractérisés que par le nombre de valeurs temporelles, qui permet de dimensionner les roues des champs.

On a ajouté quelques informations supplémentaires :

`nb_valeurs_futures` indique parmi les `nb_valeurs_temporelles` combien sont dans le futur. C'est important car ce sont les seules qui sont modifiables.

`temps_futur(int i)` indique pour chacun de ces valeurs futures quel est le temps associé.

`temps_defaut` indique sur quel temps doivent par défaut s'effectuer les opérations lorsque le temps n'est pas précisé. Pour les schémas explicites, c'est normalement le présent, et pour les schémas implicites, le futur. Dans l'idéal, tous les temps utilisés devraient être explicitement passés en paramètre des fonctions utilisant des champs, pour des raisons de lisibilité et de flexibilité.

On a mis une roue avec plusieurs valeurs temporelles sur les conditions aux limites.

Les conditions aux limites ont maintenant autant de valeurs temporelles que l'inconnue.

Lorsqu'on impose des conditions aux limites de Dirichlet sur une inconnue, l'inconnue et la condition limite sont prises au même temps, qui est passé en paramètre à `imposer_cond_lim`.

De plus, le calcul de la dérivée temporelle de la condition limite (Gpoint) est facilité. Ce calcul est fait et stocké par la méthode Gpoint() appelée par les équations après mise à jour des CLs et avant le calcul sur le pas de temps (dans `updateGivenFields`).

On a réécrit le Crank-Nicholson dans la nouvelle architecture.

Il est différencié en 2 classes :

`1. Sch_CN_iteratif` est le schéma Crank-Nicholson itératif standard. Il résoud
$u_{n+1/2} = u_n + \frac{dt}{2} f(u_{n+1/2})$
$u_{n+1} = u_n + dt.f(u_{n+1/2})$
La première étape est résolue par un point fixe :
$u^{k+1}_{n+1/2} = u_n + \frac{dt}{2} f(u^k_{n+1/2})$
Et la deuxième se simplifie en
$u^{k+1}_{n+1} = 2 * u^{k+1}_{n+1/2} - u_n)$

Ce schéma est stable jusqu'à facsec = 2. En problème couplé, il assure l'égalité des flux (de même que Euler explicite maintenant).

Le facsec est adapté automatiquement pour converger en un nombre d'itérations choisi.

`2. Sch_CN_EX_iteratif` reprend les développements de l'ancien Crank Nicholson :

* la résolution ci-dessus n'est appliquée qu'aux équations d'hydraulique.

* les autres équations font autant de pas de temps d'Euler explicite que nécessaires pour arriver à $t_{n+1/2}$ et à t_{n+1}. Ceci est répété à chaque itération de l'hydraulique (pour conserver la cohérence des conditions aux limites).

* un facteur d'amortissement (`omega`) est appliqué sur le point fixe. Avantage : empiriquement, on peut augmenter le facsec. Inconvénient : les conditions aux limites ne sont pas vraiment respectées, donc l'égalité des flux en problème couplé non plus.