Manuel d'utilisation de COSI 0.1

Emmanuel Briand

▶ **To cite this version:**

Emmanuel Briand. Manuel d'utilisation de COSI 0.1. [Rapport de recherche] RT-0245, INRIA. 2000, pp.21. inria-00069928

HAL Id: inria-00069928
https://hal.inria.fr/inria-00069928

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Manuel d'Utilisation de COSI *0.1*

Emmanuel Briand

N° 0245

Novembre 2000

THÈME 4

ISSN 0249-0803

INRIA
SOPHIA ANTIPOLIS

Manuel d'Utilisation de COSI 0.1

Emmanuel Briand*

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Caiman

Rapport technique n° 0245 — Novembre 2000 — 21 pages

Résumé : COSI (Caiman Object Solver Interface) est à la fois un environnement de programmation C++/fortran et une interface graphique d'utilisation, conçu pour fournir aux concepteurs de solveurs une aide et un standard de programmation.

Mots-clés : programmation objet, interfaçage de programmes, visualisation

* projet CAIMAN Emmanuel.Briand@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

COSI User Manual 0.1

Abstract: COSI (Caiman Object Solver Interface) provides both a C++/fortran development frame and a graphical interface. These two elements yield to solver developers an assistance and a standard for future developments.

Key-words: object oriented development, code interfacing, visualization

1 Introduction

COSI (Caiman Object Solver Interface) a été conçu pour fournir aux concepteurs de solveurs de l'équipe Caiman de l'INRIA de sophia-antipolis[1] un environnement de programmation C++/fortran (API : Application programming interface) et une interface graphique d'utilisation (GUI Graphical User Interface).

L'API est constitué de classes de maillages non-structurés 2D et 3D et de classe de base pour les solveurs. Les classes de maillage fournissent la plupart des quantités utiles aux problématiques des éléments finis/volumes finis et sont modulaires (on ne construit que les quantités nécessaires). Le concepteur d'un nouveau solveur a essentiellement pour tache d'implémenter certaines méthodes du solveur de base et éventuellement d'étendre une classe de maillage à ses besoins particuliers.

Le GUI lui permet de piloter l'exécution du solveur sur une machine locale ou distante et de visualiser l'évolution de ses variables internes. Ce schéma permet, outre une grande capacité à traiter types différents de solveurs, de fournir une interface standard vers ces solveurs et de diminuer grandement le travail de maintenance. Dans les cas où on veur adapter des solveurs déjà existant, il est facile d'utiliser les subroutines fortran initiales. Les calculs lourd s'effectuant dans ce langage, il n'y a pas de dégradation des performances et le travail d'intégration des solveurs est relativement simple. Le prix à payer pour ces deux avantages est que la structure objet des classes de maillage n'est pas strictement respectée dans la mesure où les données internes de la structure sont accessibles directement depuis l'extérieur. Pour le moment, trois solveurs ont été intégrés dans ce canevas à titre d'exemples : un solveur Euler 2D centré noeud et deux solveurs Maxwell 3D (centré noeud et centré élément).

Ce manuel va décrire la façon d'utiliser l'API et le GUI mais pas la faon de les faire évoluer (ce qui constituerait le sujet d'un « Guide du développeur »). Un complément utile sera la documentation html du code source (voir *DOC/html/index.html*), qui est jointe en annexe.

2 Installation

Pour utiliser COSI, il faut avoir installé les versions développement de PVM(3.4), QT(2.1) et VTK(3.1). Les deux derniers ne sont nécessaires que sur la machine depuis laquelle vous voudrez piloter vos solveurs. Ces logiciels sont assez faciles à trouver et à installer :

- PVM : API d'échange de données sur un réseau hétérogène[2]
- QT : API pour la création d'interfaces graphiques.[3]

1. www.inria.fr/Equipes/CAIMAN-fra.html
2. www.epm.ornl.gov/pvm/pvm_home.html
3. www.trolltech.com/products/download/freelicense/qtfree-dl.html

- VTK : API de visualisation de données 2D et 3D.[4]

Rq : On suppose que le système de l'utilisateur possède dèjà un API OpenGL (Mesa[5] ou constructeur).

COSI se présente sous la forme d'une arborescence de fichiers :

- vtkQGL : Ce répertoire contient un API permettant de faire le lien entre Qt et VTK (via l'extension OpenGL de Qt). Le source vient de Jan Ehrhardt et il est fournit avec quelques modifications (vtkQGLWidget.cpp a été modifié pour supprimer des bugs sous MESA).[6]).
- COSI : contient les répertoire représentant les différents objets du projet (GUI, maillages, solvers ...)

La marche à suivre pour compiler et installer COSI est assez simple et elle permet de le faire pour des architectures différentes depuis le même répertoire (chaque élément dépendant du système sera installé dans un sous-répertoire au nom du système d'exploitation courant). Pour l'utilisateur « normal », il suffit d'exécuter « **./configure** » puis « **make** ». Si tout se passe bien le script configure va trouver les packages et générer les fichiers de configuration. Si des packages manquent, certains élémemts ne seront pas compilés ou certaines fonctions pas activées. Configure accepte plusieurs options qui permettent d'indiquer un chemin non-standard vers un package ou de spécifier la désactivation de certaines fonctionnalités. Par défaut les librairies, les exécutables et les fichiers d'include seront installés (liens physiques) en cours de compilation dans **/usr/local/cosi (lib/ bin/ include/)** si l'utilisateur est root, et sinon sous **$HOME/cosi**. Pour lancer le GUI, il suffit d'exécuter **cosigui.exe**. En effet les chemins vers les différentes librairies dynamiques ont été inclus dans les exécutables (variable LIB_RUNTIME_DIR du **system.make**). Cela dit, si les librairies de COSI ont été déplacées « à la main », **cosigui.exe** devra être lancé via un script qui positionne la variable d'environnement LD_LIBRARY_PATH de même que **cosipilot.exe** (l'exécutable qui est lancé par le GUI pour piloter les solveurs).

3 Structure

3.1 L'API

L'API comporte un ensemble de classes de base, abstraites ou non, et de classes dérivées. La plupart sont templétées par le type des flottants pour les classes de maillages ou de solveurs, ou par le type de leur données pour les tableaux ou les paramètres.

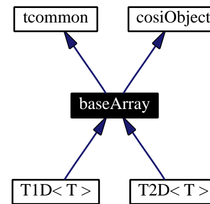
4. www.kitware.com/vtk.html
5. www.mesa3d.org
6. www.medinf.mu-luebeck.de/%7Eehrhardt/vtkQGL/vtkQGL.html

Classes de base :

cosiObject : C'est la classe parent de tous les objets du projet. Elle fournit un mécanisme de log centralisé, de gestion des erreurs et quelques fonctions de conversion. Remarquons que les procédures de log supposent que la machine de calcul (sur laquelle s'exécute **cosipilot.exe**) et la machine de contrôle (sur laquelle s'exécute **cosigui.exe**) ont accès au même espace disque (via NFS ou autre). Si ce n'est pas le cas, il suffit de lancer **cosigui.exe** dans un terminal et PVM se chargera de récupérer les messages de **cosipilot.exe** (il faudra compiler avec l'option COSI_VERBOSE).

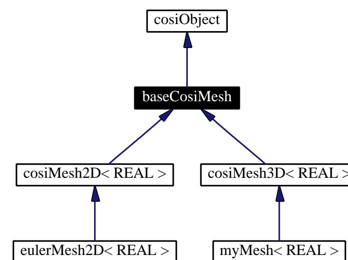
tcommon : Cette classe contient la mémoire allouée par l'ensemble des tableaux utilisés dans un exécutable.

baseArray : Classe de base pour les tableaux T1D et T2D. Elle est utilisée par baseCosiMesh (via le polymorphisme) pour gérer les tableaux optionnels des maillages.



Param : C'est la classe générique des paramètres du solveur et du GUI. Via le mécanisme du polymorphisme, elle permet de manipuler ces paramètres sans connaître leur type (lecture, sauvegarde, conversion).

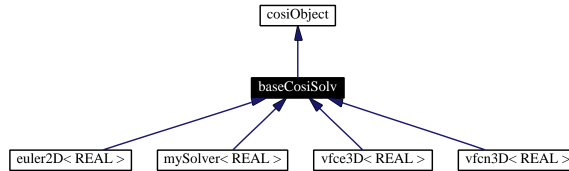
baseCosiMesh : Classe abstraite de base pour les maillages. Elle aussi est utilisée pour accéder au maillages sans connaître leurs spécificités. De plus elle fournit des méthodes génériques de lecture et de sauvegarde d'un maillage.



baseCosiSolv : Classe abstraite de base pour les solveurs. cosiPilot va contrôler le solveur via les membres de cette classe. Le concepteur d'un nouveau solveur va donc avoir à réimplémenter ses fonctions membres abstraites.

Classes dérivées :

T1D : Classe de tableaux 1D. Elle fournit des méthodes et des opérateurs permettant de manipuler ses données internes et d'en donner une vue à l'extérieur. En particulier,



elle garde trace de la mémoire occupée par ses instances et elle permet de référencer des blocs mémoire alloués à l'extérieur.

T2D : Classe de tableaux 2D. En plus des méthodes disponibles en 1D, cette classe permet aussi de trier et de faire des recherche sur des éléments de type tableau 1D (tri et recherche par ligne ou colonne).

cosiMesh2D : Classe de maillage 2D non-structuré volumes finis. En plus d'une structure minimum, elle propose la construction de différentes quantités qui peuvent être utile à un solveur volumes finis. Bien qu'aucune supposition quant à la signification physique des logiques de frontière ne soit faite, des tableaux d'indices de sommets ou d'éléments triés selon leur logique de frontière ou de coin sont accessibles.

cosiMesh3D : Classe de maillage 3D non-structuré volumes finis. Comme pour le 2D, l'utilisateur de cette classe peut demander la construction de différents tableaux

eulerMesh2D : C'est la classe de maillage spécialisée utilisée par le solveur *euler2D*. Elle hérite de *cosiMesh2D* et rajoute des traitements aux bords particuliers.

myMesh : C'est un exemple type d'évolution d'une classe de maillage (ici *cosiMesh3D*). Il illustre simplement les mécanismes d'initialisation standards que ces objets doivent implémenter.

mySolv C'est un exemple de solveur minimal. Les méthodes de *baseCosiSolv* à réimplémenter y sont définies le plus simplement possible.

euler2D Solveur des équations d'Euler bidimensionnelles centré noeud.

VFCN Solveur des équations de Maxwell tridimensionnelles centré noeud.

VFCE Solveur des équations de Maxwell tridimensionnelles centré élément.

3.2 Le GUI

Contrairement à l'API, les classes du GUI ne sont pas des templates car Qt ne les supporte pas. Par défaut le type des flottants est la double précision.

cosiComProtocol : C'est la classe dont hérite *cosiPilot* et *cosiGui*. Elle permet de gérer le protocole de communication entre ces deux objets par l'échange de messages. C'est elle qui possède toute les données échangées assurant ainsi la symétrie entre la partie exécution des solveurs et la partie contrôle de cette exécution.

cosiPilot C'est la classe qui permet de gérer le chargement et l'exécution du solveur sur la machine de calcul.

cosiGui C'est la classe qui sert d'interface graphique à l'utilisateur des solveur. Elle permet d'envoyer des ordres à cosiPilot pour charger une librairie et exécuter les

méthodes standards de baseCosiSolv. En plus de cela, elle permet de contrôler l'évolution du calcul au moyen d'une petite fenêtre de graphe 1D et d'un fenêtre de visualisation 2D ou 3D.

cosiGuiSkel C'est la classe parent de cosiGui qui décrit sa géométrie. Elle a été générée par le logiciel **qtarch** (**qtarch cosiGui.dlg**).

cosiPlotWidget C'est un des « widgets » faisant partie de cosiGui. Il sert à afficher les courbes 1D d'évolution temporelle de certains scalaires du solveur.

cosiVtkWidget C'est la classe décrivant la fenêtre dans laquelle se feront les visualisations complexes (2D ou 3D) des données des solveurs. Cet objet est basé sur le vtkQGLWidget de Jan Ehrhardt. Il permet, à partir d'objets VTK et QT, de calculer des isosurfaces (en 3D) ou des cartes de couleurs (2D) et d'interagir avec elles (zoom, déplacement, choix de variables ...).

4 Les maillages

La description complète des classes de maillages serait trop fastidieuse ici et on invite le lecteur à se référer à la documentation du code. Nous ne mentionnerons ici que leurs principes généraux d'utilisation.

Un maillage héritant de *baseCosiMesh* possède au moins ces données et ces méthodes.
Données :

- *nNode, nPoly, nCell, nNodeByPoly, nBNode ... :* Le nombre de noeuds, de polygones, de cellules, de noeuds par polygone, de noeuds frontière etc... .
- *nodeCoor :* Les coordonnées des noeuds (tableau 1D de flottant taille *nNode*).
- *polyToNode :* Les indices des noeuds composants les polygones (tableau 2D d'entiers de taille *(nNodeByPoly,nPoly)*). En général ce tableau est construit dans les maillages 2D.
- *cellToNode :* Les indices des noeuds composants les cellules (tableau 2D d'entiers de taille *(nNodeByCell,nCell)*). Évidemment ce tableau n'est construit que pour les maillages 3D.
- *supIntArray, supRealArray :* La liste des tableaux d'entiers et de flottants supportés par la classe de maillage courante. Cette liste est formée à partir d'une « map » entre des chaînes de caractères (les noms des tableaux) et un pointeur sur ces tableaux.
- *askedArray :* La liste des tableaux que ce maillage construit effectivement (un vecteur de chaînes de caractères).

Méthodes :

- *void initMesh :* Une méthode d'initialisation du maillage à partir d'un fichier descripteur et de la liste de tableau à construire.
- *void saveMesh :* Une méthode de sauvegarde du maillage (au format *cosiMesh*).

Ce *baseCosiMesh* est spécialisé en deux classes *cosiMesh2D* et *cosiMesh3D*. Les tableaux supportés par ces deux classes sont bien sûr différents mais leur façon de traiter les frontières est la même. Ils utilisent une numérotation séparée pour les objets situés sur les frontières et proposent des tableaux liant cette numérotation frontière à la numérotation interne. Par exemple, *cosiMesh2D* permet de construire les tableaux *bNodeToNode(nBNode) et nodeToBNode(nNode)*.

Ces deux classes de maillages peuvent être étendues pour des solveurs ayant des besoins particuliers (ex *eulerMesh2D* ou *myMesh*). Dans ce cas, les tableaux supplémentaires sont à rajouter à *supIntArray et supRealArray* et ils devront être construits par la méthode *void initUserMesh*. Notons que *askedArray* peut alors être rempli dans le constructeur de la nouvelle classe plutot que dans le constructeur du solveur (via la variable *baseCosiSolv : :userAskedArray*).

5 Un exemple simple de solveur

On va présenter ici la construction du solveur minimal *mySolver*. Ce solveur hérite de *baseCosiSolv*, il possède donc les données et les méthodes de celui-ci. Par convention les méthodes ou les données définies dans *baseCosiSolv* sont précédées d'un suffixe « base » tandis que les méthodes abstraites à implémenter dans le solveur sont précédées d'un suffixe « user ».

5.1 baseCosiSolv

Données :

- *string userCosiSolverVersion* : la version du solveur.
- *baseCosiMesh<REAL>*baseMesh* : un pointeur sur le maillage de base (utilisé par le GUI pour accéder au données du maillage).
- *vector<string>userAskedArray* : la liste des tableaux que le solveur va demander à la classe maillage de construire.
- *bool isCellCentered* : indication sur la structure de donnée utilisée par ce solveur ;
- *int userNVar* : nombre de variables internes du solveur (ex 5 pour un Euler compressible 3D : rho u v w p)
- *T2D<REAL>userVariables* : le tableau de taille *(nCell,userNVar)* ou *(nNode,userNVar)* contenant les variables internes.
- *vector<Param*>userMonitoredVar* : la liste des variables scalaires de dimension 0 (forcément des flottants) dont on pourra tracer l'évolution dans *cosiGui*.
- *vector<Param*>userSolvParam* : la liste des paramètres du solveur.

Les méthodes non abstraites :

- *baseCosiSolv* : Le constructeur. Il initialise les données appartenant à baseCosiSolv et remet les fichiers de log à zéro (voir *cosiObject*).

- *void baseWriteDataToFile*: cette méthode permet de sauver les variables internes du solveur (userVariables) d'une façon standard.
- *void baseReadDataFromFile*: permet de relire les données sauvées à partir de baseWriteDataToFile.

5.2 mySolver

Données :

- *myMesh<REAL>* userMesh*: c'est le pointeur sur le maillage spécifique pour ce solveur. Il hérite de *cosiMesh3D* et ajoute le tableau *uneNormaleAmoi*. Voila le code contenu dans mymesh.C.

```
template <typename REAL>
myMesh<REAL>::myMesh(){
  supRealArray["uneNormaleAMoi"]=&uneNormaleAMoi;
};
template <typename REAL>
void myMesh<REAL>::initUserMesh(){
  uneNormaleAMoi.resize(nNode);
  uneNormaleAMoi=0.;
};
```

  Si l'utilisateur veut sauver son maillage après l'avoir construit (pour gagner du temps à l'initialisation), il peut appeler la méthode *baseCosiMesh::saveMesh* à la fin de *initUserMesh*. En effet cette possibilité n'est pas proposée dans *cosiGui*.
- *int userDataType REAL aParam*: les paramètres internes du solveur.
- *REAL Hz*: une variable interne destinée à être suivie temporellement dans *cosiGui*.

Méthodes :

- *mySolver()*: le constructeur.

```
template <typename REAL>
mySolver<REAL>::mySolver(){

  /*Flag used in userInitData  to be sure that
    baseInitParam was called at least once before
    data initialization.*/
  userDataType=-1;

  userCosiSolverVersion="mySolver0.1";
  isCellCentered=false;

  addParam<int>(userSolvParam,"userDataType",userDataType);
  addParam<REAL>(userSolvParam,"aParam",aParam);

  addParam<REAL>(userMonitoredVar,"Hz",Hz);
```

```
    userAskedArray.push_back("edgeToNode");
    userAskedArray.push_back("uneNormaleAMoi");

    userMesh=NULL;
  };
```

On voit ici comment initialiser *userSolvParam*, *userMonitoredVar* et *userAskedArray*. *userMesh* est affecté de « NULL » jusqu'à ce que *userInitMesh* soit appelé.

- *void userInitMesh*

```
template <typename REAL>
void mySolver<REAL>::userInitMesh(const string& fileName) {
    // En cas d'appel multiple a init
    if(userMesh!=NULL) delete userMesh;
    userMesh=new myMesh<REAL>();

    userMesh->initMesh(fileName,userAskedArray);

    // Pour pouvoir utiliser baseMesh dans {\em cosiGui}
    // et {\em cosiPilot}
    baseMesh=userMesh;
  };
```

On prend la précaution initiale de détruire *userMesh* puisqu'il est possible qu'un utilisateur de *cosiGui* appelle plusieurs fois de suite cette méthode. *userMesh* est créé et initialisé grâce à la méthode *initMesh* de *cosiMesh3D* (héritée par myMesh). *baseMesh* pointe sur *userMesh* pour permettre à *cosiPilot* (qui ne connaît pas *myMesh*) d'accéder aux données du maillage (c'est ça le polymorphisme).

- *void userInitData*

```
template <typename REAL>
void mySolver<REAL>::userInitData(const string& fileName,
                                  REAL& time, int& iter){

  // If baseInitParam was not called before : do it now
  if(userDataType<0) baseInitParam(fileName);

  userNVar=1;
  userVariables.resize(userMesh->nNode,userNVar);

  userVariables=0;
};
```

Par précaution on appelle *baseInitParam* au début (bien que ce ne soit pas nécessaire ici et que *cosiPilot* appelle toujours *baseInitParam* avant *userInitData*).

- *REAL userSetTimeStep*

```
template <typename REAL>
```

```
REAL mySolver<REAL>::userSetTimeStep(){
  REAL dt=1.;
  return dt;
};
```

Retourne le pas de temps au pilote. Remarquons que ce solveur ne possède pas de variable « dt » mais que si les méthodes appelées dans *userTimeStep* en avaient eu besoin, c'est ici qu'il aurait fallu l'affecter.

- *bool userTimeStep*

```
template <typename REAL>
bool mySolver<REAL>::userTimeStep
     (const int& step,  const REAL& time){
  for (int i=1;i<=userMesh->nNode;i++)
    userVariables(i,1)=userMesh->uneNormaleAMoi(i);
  return true;
};
```

Fait avancer la solution d'un pas de temps et retourne qu'aucun critère interne au solveur n'a spécifié de stopper le calcul.

- *void userComputeDataForVisu*

```
template <typename REAL>
void mySolver<REAL>::userComputeDataForVisu
      (T1D<REAL>& dataForVisu, vector<string>& dataVisuLabels,
       int component){
  dataForVisu.resize(userMesh->nNode);
  T1D<REAL> tRef;
  tRef.reference(&userVariables(1,1),userMesh->nNode);
  dataForVisu=tRef;
  dataVisuLabels.clear();
  dataVisuLabels.push_back("V1");
};
```

Redimensionne le tableau *dataForVisu* et le rempli de la variable à visualiser dans *cosiGui* en fonction de la valeur de *component*. Affecte à *dataVisuLabels* la liste des variables accessibles.

- *void userSaveRestartData*

```
template <typename REAL>
void mySolver<REAL>::userSaveRestartData
        (const string& fileName, const REAL& time,const int& iter){
  writeAllParam(userSolvParam,fileName);
  string file=fileName.substr(0,fileName.find_last_of("."))+".var";
  baseWriteDataToFile(file,time,iter);
};
```

A partir d'un nom de fichier de base (normalement celui du fichier de paramètres), cette méthode sauve à la fois les paramètres courants et les variables internes du solveur. On peut noter que ces fichiers de « restart » pourront être utilisés dans

userInitData pour peu que cette méthode le prévoit (par exemple en fonction de la valeur de *userDataType*).

- *void userSaveDataForVisu*

```
template <typename REAL>
void mySolver<REAL>::userSaveDataForVisu(const string& fileName,
                                         const int&    indexNumber,
                                         const int& visuType) const
```

Cette méthode n'a pas encore été implémentée en 3D contrairement au 2D où plusieurs format de sauvegarde sont disponibles : plotmtv, vis2t (voir le source de **euler2d_savedata.hh**). Notons qu'en général *filename* va fournir le préfixe des fichiers à créer et que *indexNumber* va permettre de les numéroter.

5.3 Les fonctions de la librairie

Le solveur doit être compilé sous la forme d'une librairie dynamique que *cosiPilot* pourra charger à la demande. Une fois chargée, *cosiPilot* devra accéder au solveur lui même. Pour ce faire, la librairie doit déclarer un pointeur vers le solveur dans l'espace de nom global et un moyen de construire ce solveur. C'est le but de ce bout de code placé à la fin de **mysolver.C**

```
mySolver<double> *psolverD;
mySolver<float> *psolverS;
extern "C" {
  /*! Allocate psolverD if precision==8  or psolverS if precision==4.
    For other values of precision return false. */
  bool initLib(int precision){
    bool okInit=true;
    psolverS=NULL;
    psolverD=NULL;
    if(precision==4)psolverS=new mySolver<float>;
    else if(precision==8)psolverD=new mySolver<double>;
    else okInit=false;
    return okInit;
  }
  /*! Delete psolverD or psolverS */
  void deleteLib(){
    if(psolverS!=NULL)delete psolverS;
    if(psolverD!=NULL)delete psolverD;
    }
};
```

Cette manipulation est nécessaire pour donner accès au solveur avec des flottants en double ou en simple précision. Il faut cependant remarquer que ce choix n'est pas proposé dans *cosiGui* pour le moment car Qt ne supporte pas les templates.

6 Intégrer des subroutines fortran

C'est une procédure assez simple qui est compliquée par l'utilisation des templates. En effet on veut pouvoir choisir entre deux versions d'une routine fortran en fonction du type de *REAL*. Le seul moyen est d'avoir deux routines fortran aux noms différents, l'une recevant des arguments flottants en simple précision et l'autre en double. Le choix entre ces deux fonctions est fait au au cours de l'exécution par une conversion de pointeur de fonction.

Prenons l'exemple d'une subroutine fortran « pente », on déclare un pointeur sur une fonction recevant les même arguments :

```
typedef void (*PFPENTE)
    (int* pnNode, int* pnPoly,
     REAL* pgamma, int* pnOrdre,
     REAL* pnodeCoor, int* ppolyToNode,
     REAL* pnodeArea, int* pNOESF,
     REAL* pro, REAL* prou, REAL* prov, REAL* proe,
     REAL* pdx1, REAL* pdx2, REAL* pdx3, REAL* pdx4,
     REAL* pdy1, REAL* pdy2, REAL* pdy3, REAL* pdy4
     );
```

Puis on définit une fonction de conversion de ce pointeur vers l'une ou l'autre des routines fortran :

```
void FunctionConverter(PFPENTE& fct){
   fct=(PFPENTE) &pente4_;
   if(sizeof(REAL)==8) fct=(PFPENTE) &pente8_;
 };
```

On déclare dans l'espace de nom global le nom des routines fortran (le « _ »à la fin des noms fortran vient de la façon dont le compilateur nomme les objets).

```
extern "C" {
  // Double precision
  void pente8_(void*...);
  void pente4_(void*...);}
```

Notons que les arguments de pente8(4)_ ne sont pas importants ici puisque ces routines sont accédées via une conversion. Dans *userTimeStep* de euler2d, on appelle pente de cette façon :

```
  PFPENTE pente;
  FunctionConverter(pente);
  pente(&userMesh->nNode,&userMesh->nPoly,
        &gamma,&nOrdre,
        userMesh->nodeCoor.data(),userMesh->polyToNode.data(),
        userMesh->nodeArea.data(),userMesh->NOESF.data(),
        ro.data(), rou.data(), rov.data(), roe.data(),
```

```
        &DX(1,1), &DX(1,2), &DX(1,3), &DX(1,4),
        &DY(1,1), &DY(1,2), &DY(1,3), &DY(1,4)
        );
```

Si on avait voulu se contenter d'une version double précision il aurait suffit de sauter les deux premières étapes, de déclarer une fonction « pente » :

```
extern C{
    void pente_
       (int* nNode, int* pnPoly,
        double* pgamma, int* pnOrdre,
        double* pnodeCoor, int* ppolyToNode,
        double* pnodeArea, int* pNOESF,
        double* pro, double* prou, double* prov, double* proe,
        double* pdx1, double* pdx2, double* pdx3, double* pdx4,
        double* pdy1, double* pdy2, double* pdy3, double* pdy4);}
```

et de l'appeler telle quel.

7 Utilisation du GUI

7.1 Principes de fonctionnement

Le GUI interagit avec le solveur via l'exécutable **cosipilot.exe**. Ce dernier reçoit des demandes et des informations du GUI et lui renvoie le résultat de ces demandes. Cette correspondance est gérée par la classe parente de ces deux objets : *cosiComProtocol*.

La phase d'initialisation du calcul est constituée d'une suite de changement d'états symétriques de *cosiPilot* et de *cosiGui*. Par exemple quand le pilote reçoit le message « Load Library », il récupère le nom de cette librairie et essaie de la charger. S'il réussit, il passe dans l'état : « Attente d'un fichier de maillage » et prévient *cosiGui* de ce changement. Pour le pilote ce changement d'état se traduit par le fait qu'il accepte de recevoir de nouveaux messages de la part de *cosiGui* (voir *validMessages* dans la documentation de *cosiComProtocol*). Quand *cosiGui* reçoit ce message il change lui aussi d'état, ce qui se traduit par de nouveaux boutons accessibles, ici les boutons permettant de sélectionner le fichier de maillage. Précisons que pour *cosiGui* comme pour *cosiPilot*, ce sont les méthodes *handleMessage* et *updateState* qui gère la manipulation des messages et les actions à entreprendre quand ce message est un « Update State ».

Il est aussi possible d'initialiser le GUI d'un coup en chargant une configuration complète grâce au menu *File / Load Config*. Par défaut le fichier de configuration chargé est **.cosigui/cosiguirc**, c'est celui qui est généré automatiquement quand on ferme *cosiGui*. En voici un exemple :

```
%%%% Configuration file for Cosigui %%%%%%%
%%%% created the Thu Nov 2 17:18:02 2000 %%%%%%%
```

```
%%%% Machine Name
cervin
%%%% PVM Executable
/u/dolomites/0/caiman/ebriand/cosi/bin/SunOS/cosipilot.exe
%%%% Solver Library
/u/dolomites/0/caiman/ebriand/cosi/lib/SunOS/libeuler2d.so
%%%% Mesh File
/u/dolomites/0/caiman/ebriand/COSIPACK/DATA/EULER2D/naca800.gridCNL
%%%% Params File
/u/dolomites/0/caiman/ebriand/COSIPACK/DATA/EULER2D/naca.dataparams
%%%% Run Params File
/u/dolomites/0/caiman/ebriand/COSIPACK/DATA/EULER2D/naca.runparams
```

7.2 L'initialisation d'un solveur pas à pas

Utilisation des boutons :
Les boutons violets permettent de sélectionner un fichier dans une arborescence, les « lignes d'édition » permettent de valider l'envoi des noms de fichiers ou des paramètres vers *cosiPilot* (la validation se fait par un « retour chariot »), et les boutons rouges envoient un message « préformatté » (du type « lance le calcul » ou « pause »). A tout moment les boutons affichés peuvent être utilisés et le solveur passera dans l'état approprié. Par exemple si en cours d'exécution, l'utilisateur change le fichier de paramètres du solveur, le GUI stoppera le calcul, réinitialisera le solveur et passera en attente des paramètres de l'exécution.

Les étapes de l'initialisation :

- Le choix de *cosiPilot*. Si les librairies cosi n'ont pas été déplacées du répertoire LIB_FINAL_INSTALL_DIR donné dans **system.make**, il suffit de sélectionner l'exécutable **cosipilot.exe** correspondant à l'architecture sur laquelle vous vous lancer le calcul. Si ce n'est pas le cas il faut alors écrire un script qui positionnera la variable LD_LIBRARY_PATH sur le nouveau répertoire des librairies cosi. Vous pouvez aussi choisir sur quelle machine vous désirez lancer le calcul. Cette liste est issue de la variable d'environnement UNIXMACHINES qui répertorie les machines accessibles à PVM (protocole rsh ou ssh). Quand ce choix est fait, *cosiGui* « spawn » **cosipilot.exe** et envoie le message « Init Process » et la précision des flottants. Il lance aussi l'horloge qui permettra d'aller vérifier l'arrivée de messages en provenance de *cosiPilot* (voir *cosiGui::spawnSolverProcess* et *cosiGui::timerEvent*). Le « main » inclut dans **cosiPilot.exe** crée alors une instance de *cosiPilot* de type *double* ou *float* et lance la méthode *cosiPilot::run* :

```
template <typename REAL>
void cosiPilot<REAL>::run(){
  //Loop forever
  for(;;){
    cosiMessageId messageId=checkForMessage(guiTid);
```

```
      handleMessage(messageId);
      //If cosiGui die; cosiPilot exit
      int bufid=pvm_nrecv(-1,100);
      if(bufid!=0){pvm_exit();exit(0);}
    };
  };
```

- Le choix du solveur. est ce qui est décrit en introduction de cette section. Dans *cosiPilot*, il est géré par *loadLib* via des *dlopen()* et *dlsym()*.
- Le choix du maillage. Le type de format des fichier de maillages est spécifié par une chaîne de caractères placée en entête du fichier, elle a la forme :
  « Mesh_type :Format ». Dans *cosiMesh2D*, les formats supportés par défaut sont :
  - **CNL** : Fichier ascii contenant :
    nNode nPoly \n
    nodeCoor(2,nNode) polyToNode(nNodeByPoly,nPoly)
    bNodeLogic_tmp(2,nNode)
    voir *cosiMesh2D<REAL> : :readCNLMesh*.
  - **cosimesh** : Ce format est généré par *saveMesh* et permet de sauvegarder tout ou partie des tableaux construits par le maillage,
    voir *cosiMesh2D<REAL> : :readCosiMesh*.
  - **Build_from_param** : Fichier ascii contenant :
    nx ny \n
    X0 Y0 Xl Yl \n
    iStruct \n
    logic1 logic2 logic3 logic4
    voir *cosiMesh2D<REAL> : :buildMeshFromParam*.

  Dans *cosiMesh3D*, les formats supportés par défaut sont :
  - **CNL** : Fichier ascii contenant :
    nNode nCell nBPoly \n
    nodeCoor(3,nNode) cellToNode(nNodeByCell,nCell)
    bPolyLogic(nBPoly) bPolyToNode(nNodeByPoly,nBPoly)
    voir *cosiMesh3D<REAL> : :readCNLMesh*.
  - **cosimesh** : voir *cosiMesh3D<REAL> : :readCosiMesh*.
- Le choix des paramètres du solveur. La syntaxe de ces fichiers est un peu particulière car elle permet de lire les trois champs qui composent un *Param* de façon générique (nom, valeur et commentaire). Chaque *Param* est défini par la ligne suivante :

  ```
  <<N> name > <<V> value > <<C> comment >
  ```
- Le choix des paramètres de l'exécution. La saisie de ce fichier est facultative ; si le nom fourni dans la ligne d'édition ne correspond à aucun fichier, les paramètres par défauts sont générés. Notons que la méthode *baseCosiSolv : :userInitData* est appelé seulement après la lecture de ce fichier (et lors du redémarrage d'un solveur déjà lancé) et non lorsque l'utilisateur modifie à la main les paramètres dans le GUI. Ceci implique que si le concepteur du solveur n'a pas prévu que ces paramètres pouvaient être modifiés en cours d'exécution (dans la méthode *userTi-*

meStep), des inconsistances peuvent apparaître. Ci-dessous un exemple de fichier « runParams » :

```
%%%itermax
1000
%%%timeMax
100.
%%%iterFreqSave
1000
%%%iterStartSave
100
%%%visuSaveType
1
%%%evolWidgetRefreshRate
3
%%%visuRefreshRate
50
```

iterFreqSave permet de choisir la fréquence de sauvegarde des fichiers de visu, *visuSaveType* le type de fichiers de visu (cf *baseCosiSolv : :userSaveDataForVisu*), *iterStartSave* l'itération à partir de laquelle la sauvegarde commencera. *evolWidgetRefreshRate* donne la fréquence de rafraîchissement de *cosiPlotWidget* (graphe 1D) et *visuRefreshRate* celle de *cosiVtkWidget* (visu 2D-3D). Ces paramètres sont repartis en 2 groupes, les dynamiques qui sont modifiés pendant l'exécution par *cosiPilot* (*iter, time*) et les statiques qui sont modifiables par l'utilisateur. En cas de modification ces derniers sont envoyés à *cosiPilot* où ils conditionneront la fréquence d'envois des données ou leur sauvegarde (cf *cosiPilot<REAL> : :loopOneStep()*).

7.3 Contrôle d'une exécution

Le GUI propose ces différents contrôles :

- Start : passe dans l'état « stateLooping ». Si le solveur avait déjà été démarré, cosiPilot exécute *userInitData*, puis en l'absence de message boucle sur *userTimeStep*. Cette boucle continuera jusqu'à ce que l'utilisateur passe dans l'état « statePaused » ou que *cosiPilot* passe lui même dans l'état « stateFinished » (si *userTimeStep* retourne « false » ou si *iterMax* ou *timeMax* sont dépassés).
- Pause : passe dans l'état « statePaused ».
- Kill : Tue le processus de *cosiPilot*. *cosiGui* reçoit alors un message envoyer par PVM lui indiquant que *cosiPilot* est mort et il passe dans l'état « stateWaitSpawn ».
- Continue : Repasse dans l'état « stateLooping ».
- Save Data : Propose à l'utilisateur de sauver ses données pour les visualiser plutard. Si *iterFreqSave* est différent de 0, cette manoeuvre est interdite pour préserver la périodicité des sauvegardes.

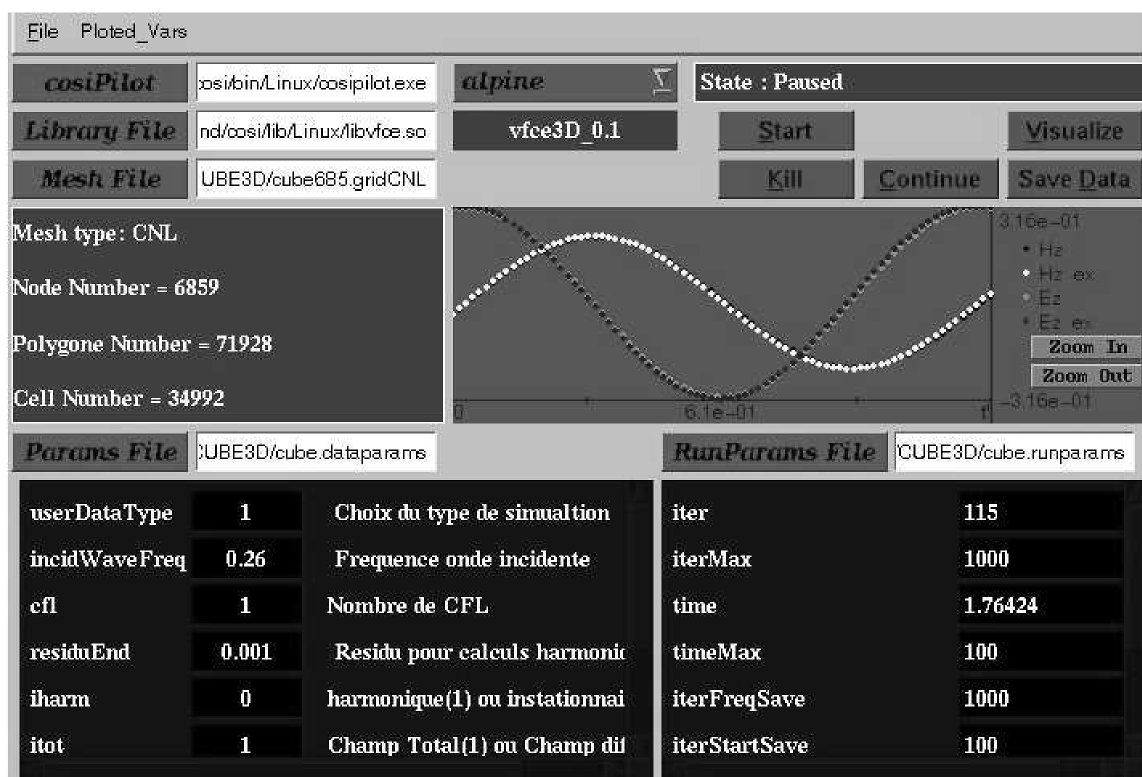


FIG. 1 – *cosiGui contrôlant un solveur maxwell 3D*

– Visualize : Envoie un message vers *cosiPilot* lui demandant les données nécessaire à initialisation d'une visualisation 2D ou 3D (le maillage et un tableau de scalaire). Quand le dernier de ces deux paquets de données est reçut par *cosiGui* (voir *cosiGui : :handleMessage*), la méthode *cosiGui : :updateVisu* est appelée. Celle-ci crée alors une instance de *cosiVtkWidget* qui construit une structure de donnée VTK, puis appelle *cosiVtkWidget : :updateData*. C'est cette dernière méthode qui affichera la visualisation demandée.

7.4 cosiPlotWidget

Ce « widget »,intégré au GUI, permet tracer les scalaires choisis dans le menu « Ploted_Vars » en fonction d'un temps relatif (le premier point représenté a toujours $t' = 0$). Ces scalaires sont ceux du tableaux *baseCosiSolv : :userMonitoredVar*. Le nombre de points représentés varie suivant le zoom mais ne dépasse pas 200. Si plusieurs scalaires sont tracés, l'échelle verticale s'adapte au minimum et au maximum global.

7.5 cosiVtkWidget

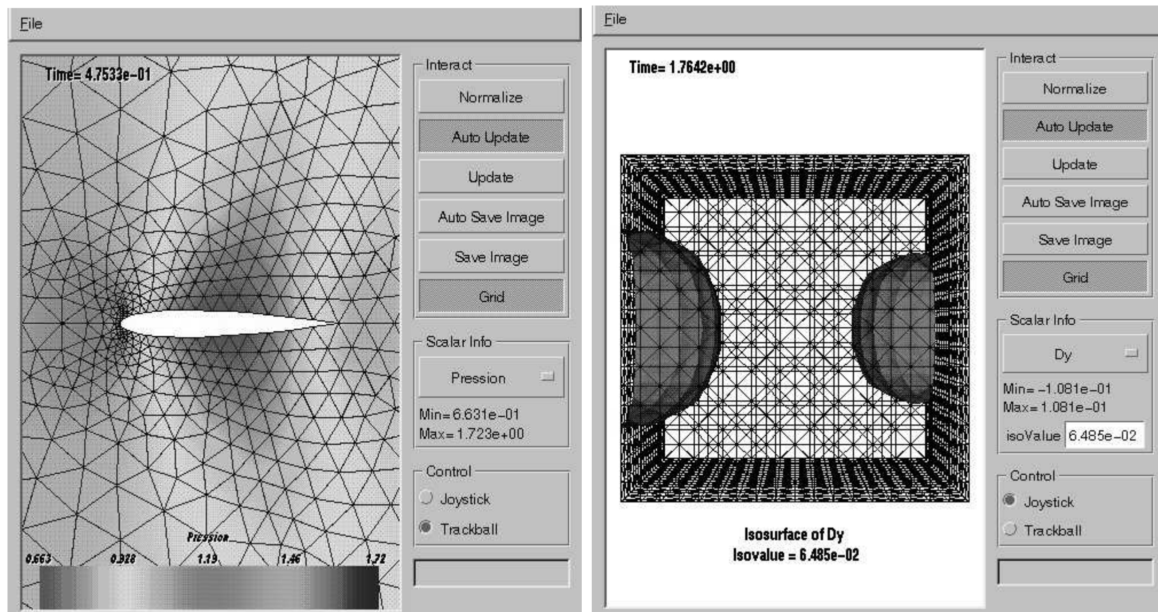


FIG. 2 – *cosiVtkWidget représentant des données 2D et 3D.*

Ce « widget » apparaît dans une fenêtre séparée et a ses propres boutons de commande. La fonction *cosiVtkWidget : :updateVisu* est appelée chaque fois que *cosiGui* reçoit les données à visualiser. Après que les calculs de représentation sont terminés, *cosiVtkWidget* renvoie vers *cosiPilot* un message lui précisant qu'il est disponible pour traiter de nouvelles données. S'il arrive que *cosiPilot* ait à envoyer des données et que *cosiVtkWid-*

get n'ait pas fini de traiter les précédentes, l'envoi est annulé, ceci pour éviter de remplir le canal de communication.

La souris peut être utilisée pour modifier la position de la caméra autour de l'objet visualisé : le bouton de gauche pour les rotations, le bouton du centre pour les translations dans le plan de la fenêtre et le bouton de droite pour les zooms.

Notons que les données du *cosiVtkWidget* ne sont pas détruites quand la fenêtre est fermée via le bouton du cadre mais seulement quand on utilise la fonction « Exit » du menu « File ».

Voici une brève description des différents boutons fournis par *cosiVtkWidget* :

- « Reset Camera » replace la caméra dans sa position d'origine lui permettant de montrer tout l'objet visualisé.
- « Auto Update », quand il est activé, indique que *cosiPilot* enverra automatiquement les données à visualiser tous les « visuRefreshRate ».
- « Update » envoie le message demandant les données à visualiser à *cosiPilot*.
- « Auto Save Image », quand il est activé, indique que l'image représentée dans la fenêtre sera sauvegardée (son nom étant indexé par le nombre de sauvegardes déjà effectuées) au format ppm à chaque remise à jour. Notons que quand cette fonction est active, le comportement de *cosiPilot* face aux demandes de données pour la visu change. En effet, au lieu d'annuler l'envoi en cas de retard de traitement des données par *cosiVtkWidget*, *cosiPilot* pause le calcul jusqu'à ce que *cosiVtkWidget* soit à nouveau disponible. Cette mesure est prise pour avoir des séries d'images avec un échantillonnage fixe.
- « Save Image » permet de sauver l'image courante (l'indexation des noms d'images se poursuivant).
- « Grid » permet d'activer ou de désactiver la visualisation de la grille.
- « Scalar info » permet de choisir la variable à visualiser. Dans le cas 3D, il permet aussi de choisir la valeur de l'isosurface. Notons que cette valeur est positionnée par défaut à $max - 0.2 * (max - min)$ mais qu'elle devient fixe quand l'utilisateur la modifie (tant qu'on ne change pas la variable).
- « Control » permet de choisir les modes d'actions de la souris dans la fenêtre graphique. En mode « Joystick » la portée de l'action est déterminée par le temps des clicks et la distance du pointeur au centre de la fenêtre. En mode « Trackball », c'est le déplacement du pointeur clické qui conditionne l'action.

Table des matières

1 Introduction 3

2 Installation 3

3 Structure 4

3.1 L'API 4

3.2 Le GUI 6

4 Les maillages 7

5 Un exemple simple de solveur 8

5.1 baseCosiSolv 8

5.2 mySolver 9

5.3 Les fonctions de la librairie 12

6 Intégrer des subroutines fortran 13

7 Utilisation du GUI 14

7.1 Principes de fonctionnement 14

7.2 L'initialisation d'un solveur pas à pas 15

7.3 Contrôle d'une exécution 17

7.4 cosiPlotWidget 19

7.5 cosiVtkWidget 19

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803