Rapport d'avancement de thèse
Bilan final
Juin 2013

Xavier Faure

LIRIS, équipe SAARA, projet ETOILE
UMR5205, F-69622, France
Domaine scientifique de la Doua,
23-25 Av. Pierre de Coubertin, 69100 Villeurbanne Cedex
xavier.faure@liris.cnrs.fr

Encadrants : Fabrice Jaillet, Florence Zara et Jean-Michel Moreau

Chapitre 1

Implémentation dans la librairie de simulation *SOFA*

1.1 Introduction

L'objectif général de nos travaux de recherche est de trouver les stratégies qui nous permettent de converger vers une application temps réel. Pour l'instant, nous avons vu que la *Méthode des Masse-Tenseurs* est une méthode qui offre un bon compromis temps de calcul — précision par ailleurs encore amélioré par l'extraction de données constantes calculées à l'initialisation de la simulation. De plus, grâce au calcul formel, nous avons généralisé cette méthode pour tout type d'éléments et une loi de comportement linéaire *Hooke* ou non-linéaire *Saint-Venant Kirchhoff*. Nous avons parallélisé notre code sur *GPU* de manière générique pour le calcul des forces internes.
Le but de cette partie est de comparer nos travaux avec l'existant. Pour cela, nous avons choisi d'intégrer la *Méthode des Masse-Tenseurs* que nous avons développée à la librairie de simulation temps réel *SOFA* [1, 5]. La librairie *SOFA* est une librairie de simulation employée dans la recherche académique depuis une dizaine d'années. Beaucoup de chercheurs et d'industriels estiment que *SOFA* est la librairie de simulation temps réel *Open Source* la plus utilisé en mécanique des objets déformables et la plus proche de ce que nous souhaitons pour comparer nos résultats.
Dans ce chapitre, nous allons présenter la librairie de simulation temps réel *SOFA* et expliquer l'objectif des nouveaux composants que nous avons construits. À noter que pour comparer nos résultats, nous avons sélectionné des tests classiques de traction, de flexion, de torsion, ... sur une barre paramétrable. La Figure 1.1 montre un exemple de barre formée uniquement de tétaèdres. Dans un second temps, nous présentons notre générateur de barres hybrides, mélangeant tout type de loi de comportement et tout type d'éléments.

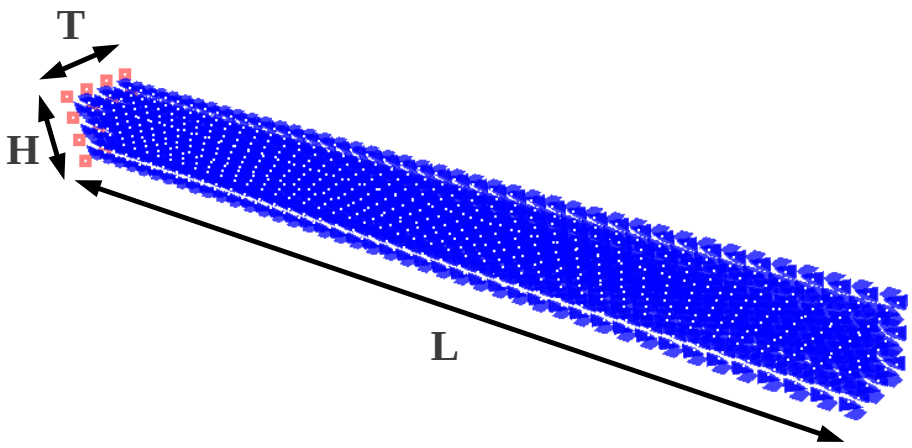


FIGURE 1.1 – Barre type utilisée pour les tests.

Grâce au calcul formel, nous sommes capables de simuler jusqu'à une cinquantaine d'éléments différents. Ces derniers peuvent être associés à deux types de loi de comportement. Les équations générées peuvent être découpées de manière différente et exécutées sur deux types de processeur, le *GPU* ou le *CPU*. Pour traiter cet ensemble volumineux de tests, nous a construit une plateforme. Avec celle-ci, nous pouvons :

- nous interfacer avec une base de données pour faciliter la classification des tests et des résultats pour une analyse efficace.
- nous baser sur la librairie *SOFA*. En effet, se baser sur une librairie comporte un certain nombre d'avantages :
  - réutiliser les composants de la librairie déjà existants.
  - se comparer rapidement avec d'autres lois de comportement ou d'autres concepts de réalisation.
  - pérenniser le code. D'autres personnes peuvent utiliser les outils que nous développons.
- gérer des maillages hybrides mélangeant plusieurs lois de comportement et plusieurs types d'éléments.
- générer des groupes de simulations pour réaliser plusieurs tests en une seule commande.
- générer des graphes de résultats.

Par conséquent, dans un troisième temps, nous expliquons les fonctionnalités de notre plateforme de génération et d'exécution de groupes de simulations pour la réalisation d'ensemble de tests. Enfin, nous montrons l'intérêt d'avoir stocké les résultats dans une base de données et nous détaillons les fonctionnalités de notre outils d'analyse de résultats pour la génération automatique de graphes de comparaison.

1.2 Librairie de simulation

Dans le cadre de la recherche, la librairie *SOFA* est bien appropriée pour la simulation temps réel des objets déformables.

1.2.1 Librairie *SOFA*

SOFA est une librairie de simulation *Open Source* [1]. Le but premier de cette librairie est la simulation temps réel et en particulier la simulation médicale. Elle est principalement utilisée par la communauté recherche pour l'aide à la création de nouveaux algorithmes, mais aussi pour la réalisation de simulateur en partenariat avec des entreprises. Basée sur une architecture logicielle avancée, elle permet de :

- créer des simulations complexes et évoluées en combinant des nouveaux algorithmes et des algorithmes déjà existants dans *SOFA*.
- modifier un grand nombre de paramètres de la simulation en éditant un fichier `xml`. Les modifications des paramètres peuvent avoir lieu pendant la simulation. Cela peut être la loi de comportement, la représentation surfacique, la méthode d'intégration, les contraintes, les algorithmes de collision, ...
- construire des modèles complexes [3] à partir de brique simple de base en utilisant le principe de graphe de scène.
- simuler efficacement l'interaction dynamique entre des objets.
- réutiliser et comparer une grande variété de méthodes.

Actuellement, *SOFA* est développé par 3 équipes de l'INRIA : *SHACRA*, *IMAGINE* et *ASCLEPIOS*. Ce projet est également en liens avec les groupes CIMIT Sim, ETH Zurich et CSIRO. Récemment, une entreprise est née : *InSimo*. Elle vend des plugins compatibles avec le noyau de *SOFA*, aide aussi à la maintenance du noyau et lui donne une meilleur visibilité.
La librairie *SOFA* est découpée conceptuellement en deux parties [7] :

- une base logicielle avec une architecture avancée qui fait le cœur de cette librairie.
- un ensemble de modules de base permettant de définir les fonctionnalités d'une librairie de simulation.

Les composants sont classés selon des catégories : méthode d'intégration, masse, calcul de forces internes, collision, ... Un fichier `xml` permet de construire une simulation. Déclarer une nouvelle balise avec des arguments permet d'utiliser un composant. Les composants sont organisés dans un graphe pour pouvoir créer des modélisations complexes à partir de brique simple.
Un même objet peut être modélisé de plusieurs manières simultanément :

- la modélisation de la collision. Cet ensemble de modules de base permet de gérer les interactions entre les objets modélisés.

- la visualisation. Chaque objet peut avoir une dizaine de représentations visuelles différentes allant de la visualisation simple des nœuds jusqu'au recalage de surface sur un modèle 3D.
- le calcul des forces internes. Chaque vecteur force peut être représentée par une flèche.

Les visualisations peuvent être masquées à tout moment grâce à l'édition en ligne de *SOFA*.

1.2.2 Composants *SOFA* existants

Nous allons décrire les composants principaux de *SOFA* au travers d'un exemple qui vise à simuler une barre encastrée sur un mur. Cette barre est soumise à la gravité.

MeshTopology. Ce composant contient les nœuds du maillage. Il organise les nœuds du maillage en élément, en face et en arête. Pour l'import 2D, le format `obj` est le format standard. Les formats `vtk` et `mesh` sont deux autres possibilités. Pour l'import 3D, le format `gmsh` est le format standard. Les formats `xsp`, `mesh` et `cgal` sont trois autres possibilités. Les topologies acceptées par ce module ne sont pas hybrides. Ainsi, pour *SOFA*, un maillage est un ensemble d'éléments de même type. Nous utilisons ce composant uniquement pour charger les nœuds du maillage. Ces nœuds sont communiqués par la suite au composant `MechanicalObject`. Le nom du fichier d'entrée est le seul argument de ce composant comme le montre le code 1 écrit en `xml`.

Code 1 Code `xml` pour ajouter la topologie.

```
<!-- les noeuds du maillage et les vecteurs d'état -->
<MeshTopology name="mesh" fileTopology="fileName.obj"/>
<MechanicalObject name="dof" />
```

MechanicalObject. Ce composant gère les variables liées aux calculs de la mécanique. Il est complètement lié à `MeshTopology`. C'est lui qui crée les vecteurs d'état comme le vecteur force, le vecteur vitesse, le vecteur position et tous les vecteurs dont la dimension est égale au nombre de nœuds dans le maillage. Le seul attribut de ce composant est *name* comme le montre le code 1 écrit en `xml`. C'est un composant simple mais qui contient les données les plus importantes.

CGLinearSolver. Ce composant permet de résoudre un système S de la forme $AX = B$ avec la méthode du gradient conjugé. Il est nécessaire de résoudre un tel système lorsque nous utilisons une méthode d'intégration numérique avec la méthode d'Euler implicite. L'explication de cette algorithme ne fait pas partie de l'objectif de cette thèse. Toutefois, l'annexe **??** le présente pour comprendre les paramètres de l'algorithme. Hadrien Courtecuisse [2] explique cet algorithme en détails ainsi que sa parallélisation sur *GPU*. Les paramètres que nous devons régler sont :

- `iterations` : le nombre maximum d'itérations de l'algorithme du gradient conjugué.
- `tolerance` : premier seuil comparé au rapport entre la norme de `b0` et la norme de `b`. Si ce rapport est inférieur à `tolerance`, la variable `x` courante est considérée comme solution du système `S`.
- `threshold` : deuxième seuil comparé au produit scalaire entre q_n et q_{n+1}. Si ce produit est inférieur à `threshold`, la variable `x` courante est considérée comme solution du système `S`.

Le code 2 écrit en `xml` permet d'ajouter le composant gradient conjugué.

Code 2 Code `xml` pour ajouter le composant gradient conjugué.

```
<!-- Gradient conjugué pour résoudre AX = B -->
<CGLinearSolver name="CG" iterations="100"
  tolerance="1e-6" threshold="1e-6"/>
```

EulerExplicitSolver - EESolver. Ce composant permet de résoudre un pas de la simulation avec le schéma d'intégration Euler explicite. Quand le paramètre `symplectic` vaut 1, la méthode d'intégration est semi-implicite sinon il est explicite comme le montre le code 3. Il est rarement utilisé étant réputé pour être instable et nécessitant des pas de temps très petits. Cependant, il permet de comprendre le principe de base de la simulation sans introduire de notions complexes.

Code 3 Code `xml` pour ajouter la méthode d'intégration Euler explicite.

```
<!-- Méthode d'intégration Euler semi implicite -->
<EESolver name="EE" symplectic="1" />
```

EulerImplicitSolver - EISolver. Ce composant permet de résoudre un pas de la simulation avec le principe d'Euler implicite. C'est une méthode d'intégration performante et précise. Par contre, le fait d'utiliser Euler implicite impose d'avoir calculer la matrice des dérivées partielles des forces internes. Une des grandes avancées de cette thèse est de permettre l'utilisation avec la *Méthode des Masse-Tenseurs* la méthode d'intégration Euler implicite quelque soit la loi de comportement utilisée, quelque soit l'élément et quelque soit le processeur, *GPU* ou *CPU*. Le paramètre `vdamping` permet de mettre un amortissement sur le mouvement pour avoir une convergence plus rapide comme le montre le code 4. Toutefois, si ce paramètre est trop élevé, alors la convergence est plus lente.

Code 4 Code `xml` pour ajouter la méthode d'intégration Euler implicite.

```
<!-- Méthode d'intégration Euler implicite -->
<EISolver name="EI" vdamping="5" />
```

PartialFixedConstraint. Ce composant permet de fixer des nœuds selon une direction. Pour exemple, pour un maillage 2D et pour éviter que les nœuds 0, 2 et 4 se déplace dans la direction des abscisses, l'argument `fixedDirections` vaut "1 0" et l'argument `indices` vaut "0 2 4" comme le montre le code 5.

Code 5 Ajouter une contrainte en x pour les nœuds 0, 2 et 4.

```
<!-- boque les noeuds 0 2 4 en x -->
<PartialFixedConstraint name="PFC"
indices="0 2 4" fixedDirections="1 0" />
```

UniformMass. Ce compostant permet de spécifier la masse totale du maillage. Cette masse est répartie uniformément sur les nœuds du maillage. L'argument `totalmass` correspond à la masse totale du maillage comme le montre le code 6.

PartialLinearMovementConstraint. Ce composant permet d'imposer un mouvement linéaire dans une direction pour un ensemble de nœuds. Ce composant nous montre qu'il est possible de déclarer les attributs du composant de manière hiérarchique comme le présente le code 7 écrit en `xml`. Cela peut être utile lorsque les attributs sont compliqués. Cet exemple décrit un mouvement linéaire de 40 centimètres en 1 seconde des points 3 et 4. Ces points restent dans cette position pendant 1000 secondes, du temps 1.0 au temps 1001. Après le temps 1001, les points 3 et 4 reviennent à leur position initiale.

1.2.3 Composants développés dans le cadre de nos travaux

Nous avons développer différents composants que nous avons jugés important de mettre en œuvre pour notre plateforme de génération de simulations et pour pouvoir gérer les maillages hybrides. Dans un premier temps, nous avons créé le composant `VariableTimer` pour évaluer le temps passé dans des parties spécifiques du code et pouvoir ajouter directement les résultats dans notre base données. Dans un deuxième temps, nous avons créé le composant `MixTopology` pour pouvoir gérer les maillages mixtes. Dans un troisième temps, nous avons créé les composants `TM` et `TMG` pour calculer les force internes et leurs dérivées avec la *Méthode des Masse-Tenseurs* sur *CPU* et *GPU*. Enfin, nous avons créé le composant `TrackRunSofa` pour suivre la simulation et arrêter la simulation selon des paramètres lorsqu'elle est considérée comme étant stable.

Code 6 Ajouter une contrainte en x pour les nœuds 0, 2 et 4.

```
    <!-- Uniform mass component -->
    <UniformMass name="mass" totalmass="1000"/>
```

Code 7 Ajouter une contrainte de mouvement en x pour les nœuds 3 et 4.

```
  <!-- Move on x 3 and 4 onto 40 centimeters -->
  <PartialLinearMovementConstraint movedDirections="1 0" indices="3 4" >
    <Attribute type="keyTimes">
    <Data value="0    1.0    1001" /></Attribute>
    <Attribute type="movements">
    <Data value="0 0  0.4 0  0.4 0"/></Attribute>
  </PartialLinearMovementConstraint>
```

VariableTimer. Ce composant permet de communiquer aux différents composants de *SOFA* le temps qui s'écoule entre chaque pas de simulation et à l'intérieur des fonctions qui calcule les forces internes et leurs dérivées. Avec la librairie *SOFA*, pour communiquer entre les composants, il faut passer par les attributs contenus dans les balises `xml` du fichier de configuration de la simulation. Prenons l'exemple où nous souhaitons partager la variable `listChrono` qui est un objet de classe `ListChrono` au sein $n+1$ composants *SOFA* nommés de C_0 à C_n. Un pointeur sur la variable à partager est créé pour que chaque composant est accès à sa lecture et à son écriture.
Le premier composant C_0 est celui qui crée cette variable. Dans cette conception, nous pouvons considérer ce composant comme le serveur et les autres comme des clients. Cette création nécessite :

- la définition des fonctions amies `operator >>` et `operator <<` respectivement des classes `std::istream` et `std::ostream` comme le montre le code 8. Pour que le code *SOFA* fonctionne, ces deux opérateurs doivent être surchargés.
- pour le composant C_0 :
  - la déclaration du code 9 de cette variable dans le *header* de ce composant qui permet d'encapsuler le pointeur pour le communiquer aux autres composants.
  - l'augmentation de la portée de cette variable aux autres composants comme le montre le code 10.
- pour chaque composant client C_i avec $i \in \{1, .., n\}$:
  - la déclaration du code 9 de cette variable dans le *header* du composant C_i. Cette déclaration prépare la connexion entre le serveur et le client.
  - l'ouverture de cette variable sur les autres composants comme le montre le code 10. Cela correspond à augmenter la portée de cette variable pour atteindre celle du serveur.

Code 8 Définition des fonctions amies *ostream* et *istream*.

```
friend std::istream & operator >> (std::istream & c, ListChrono * a)
  { //Code to read a  list  chrono;
    ...; return c; }
friend std::ostream & operator << (std::ostream & c, const ListChrono * a )
  { //Code to write a  list  chrono;
    ...; return c; }
```

Désormais, tout est en place pour établir la communication de la variable `listChrono` entre les compostants grâce au fichier `xml` comme le montre le code 11. Sur ce fichier `xml`, nous notons que :

- le composant C_0 n'a pas besoin d'avoir la variable `listChrono` en argument. En effet, pour le composant C_0 qui est le serveur, cette variable est en sortie. Pour tous les autres composants, cette variable est en entrée.

Code 9 Déclaration de listChrono dans C_i.

```
    //The chrono variable
    Data<ListChrono *> listChrono;
```

Code 10 Déclaration de listChrono dans C_0.

```
    //Increase the scope of listChrono
    listChrono(initData(&listChrono, "listChrono", "The list of chrono every where in the sofa program"))
```

- l'ordre est important. Le serveur doit être créé en premier. La ligne correspondante doit être placée avant dans le fichier `xml`. Les autres composants sont placés après dans n'importe quel ordre puisqu'ils n'ont pas de lien entre eux.

Code 11 Déclaration des $n+1$ composants communiquant `listChrono`.

```
    <!-- Le serveur -->
    <C0 name="nameC0" />

    <!-- C1 -->
    <C1 listChrono="@nameC0.listChrono" />
    ...
    <!-- Cn -->
    <Cn listChrono="@nameC0.listChrono" />
```

MixTopology. Le composant existant `MeshTopology` de la librairie *SOFA* ne permet pas de gérer les maillages hybrides en terme de loi de comportement et de type d'éléments. Mais par contre, nous souhaitons tout de même utiliser ce composant pour maintenir le découpage entre la topologie et le composant `MechanicalObject` de *SOFA*. Ainsi, nous avons organisé les informations en trois composants, pour les particules, pour les « vecteurs d'état » et pour la topologie hybride :

- `MeshTopology` : Ce composant permetle chargement des nœuds du maillage via l'attribut `fileTopology` qui contient le chemin vers les coordonnées des particules.
- `MechanicalObject` : Nous n'avons pas modifié le rôle de ce composant. À partir des nœuds de `MeshTopology`, il crée les « vecteurs d'état » de la simulation.
- `MixTopology` : Notre nouveau composant permet de récupérer la topologie du maillage dans un premier fichier. Dans un second, nous récupérons la liste des éléments du maillage. Chaque élément est associé à son association `LED`, *Loi* de comportement, *Element* et *Decoupage*, la manière de découper les équations générées.

Prenons l'exemple d'un maillage 2D constitué de 2 triangles comme le présente la Figure 1.2. Comme pour tout maillage, deux fichiers sont nécessaires :

- `maillage2D.element` décrit dans le code 12 : ce fichier commence par une ligne contenant la lettre `'s'` suivi d'un espace et du nombre d'éléments. Pour chaque élément, nous avons :
  - le nombre de nœuds pour l'élément courant.
  - la liste des indices des nœuds.
- `maillage2D.info` décrit dans le code 13 : ce fichier commence par une ligne contenant la chaine `"nbDim"` suivi d'un espace et du nombre de dimension. Ce nombre de dimension est 2 ou 3 respectivement en 2D ou en 3D. La ligne suivante donne la chaine de caractère `$ELEM`, comme les balises dans le format `gmsh` qui

indique le début de la définition des éléments. La chaine suivante est le nombre d'éléments. Pour chaque élément, nous avons :

- l'indice de l'élément.
- le nom de la `Loi` de comportement de l'élément.
- le nom de l'`Element`.
- le nom du composant `Decoupage`.

Le fichier fini par la balise fermante `$ENDELEM`.

Code 12 Fichier `maillage2D.element`.

```
s 2
3 0 1 3
3 0 3 2
```

Code 13 Fichier `maillage2D.info`.

```
nbDim 2
$ELEM
2
0 Hooke Triangle DecoupageNon
1 Hooke Triangle DecoupageNon
$ENDELM
```

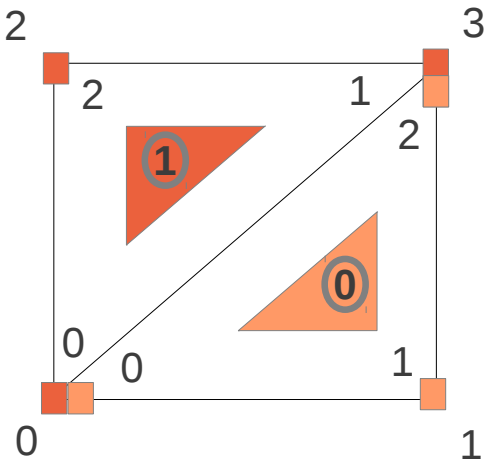


FIGURE 1.2 – Exemple de maillage 2D.

TM / TMG. `TM` correspond à la version *CPU* et `TMG` à la version *GPU* de ce composant. Ces deux compostants intègrent les classes contenant les classes contenant les équations générées représentant les forces internes et de leurs dérivées.
Dans la librairie *SOFA*, chaque simulation doit contenir un composant qui calcule les forces internes et leurs dérivées. Cela correspond au modèle physique utilisé basé sur la *Mécanique des Milieux Continus*. Si un utilisateur

de *SOFA* veut définir son propre modèle physique, il doit hériter du composant `ForceField` qui contient les fonctions de base pour le calcul des forces internes. La fonction `addForce` permet de calculer les forces internes et la fonction `addDForce` permet de calculer le produit entre la dérivée des forces internes et un vecteur `v` en paramètre. Les prototypes des méthodes `addForce` et `addDForce` présentés dans le code 14 sont aussi imposés par *SOFA*. Le paramètre commun est `mparams`. Grâce à cet objet, nous pouvons accéder à l'ensemble des « vecteurs d'état ». Dans le cadre de `addDForce`, il nous permet également de récupérer le facteur multiplicatif `h` ou $-h^2$ selon si le calcul correspond à $-h^2 \frac{\partial F}{\partial U} V(t)$ ou à $h \frac{\partial F}{\partial U} \Delta V$. Les autres paramètres sont spécifiques à chacune des deux méthodes :

- `addForce` : `d_f` vecteur contenant les forces aux nœuds. C'est un paramètre utilisé en écriture. `d_x` contient la position courante des nœuds. C'est un paramètre utilisé en lecture. Pour certains modèles physiques, le calcul des forces internes dépend de la vitesse. Dans ce cas, le paramètre `d_v` est utile. Dans notre contexte, il n'est pas nécessaire.
- `addDForce` : `d_df` vecteur contenant le produit entre la matrice des dérivées partielles des forces internes et un vecteur `v`. C'est un paramètre utilisé en écriture. `d_dx` contient la valeur courante du vecteur `v` aux nœuds. C'est un paramètre utilisé en lecture. Dans notre contexte, nous avons besoin de la position courantes des nœuds. Nous la récupérons grâce au paramètre `mparams`.

Code 14 Prototype des fonctions *addForce* et *addDForce*.

```
template<class DataTypes>
void TM<DataTypes>::addForce(const core::MechanicalParams* mparams,
DataVecDeriv & d_f, const DataVecCoord & d_x, const DataVecDeriv & d_v);

template <class DataTypes>
void TM<DataTypes>::addDForce(const core::MechanicalParams* mparams,
DataVecDeriv& d_df, const DataVecDeriv & d_dx);
```

Dans la phase d'initialisation des deux composants, nous calculons les expressions constantes regroupantes pour chaque élément. Cette phase est hors boucle de simulation. Grâce au calcul formel, nous avons extrait les expressions constantes regroupantes et nous les avons intégrées dans des classes en fonction de la loi de comportement, du type d'éléments et de la manière dont les équations sont découpées. Pour chaque élément, en fonction de son association `LED` associée, nous calculons et stockons la valeur des expressions constantes regroupantes correspondantes. Par exemple, la ligne `listECR["Hooke"]["Triangle"]["DecoupageNon"][0]` contient les `ECR` qui permet de calculer les expressions constantes regroupantes de la loi de comportement *Hooke*, pour l'élément triangle 3 nœuds et pour le `fragment` 0 du composant `DecoupageNon`. Pour chaque élément E, l'initialisation consiste à calculer les expressions `ECR` correspondantes à l'association `LED` de E comme le suggère le code 15. Une fois les données constantes calculées, la phase d'initialisation est terminée.

Code 15 Initialisation des données de l'élément E avec une association `LED`.

```
//For each fragment splitForce
for (s=0;s<nbS; s++)
{
  //Run Function Compute constant data (E->d.D)
  listECR[L][E][D][s]->compute(E->d,E->d.D[s]);
}
```

La phase de simulation est orchestré par le composant la méthode d'intégration de la boucle de simulation. Dans le cas de la méthode d'intégration Euler implicite, le système à résoudre à chaque pas de simulation est :

$$\underbrace{\left(M - h\frac{\partial F}{\partial V} - h^2\frac{\partial F}{\partial U}\right)}_{A} \underbrace{\overrightarrow{\Delta V}}_{x} = \underbrace{h(\overrightarrow{F_t} + h\frac{\partial F}{\partial U}\overrightarrow{V_t})}_{b} \tag{1.1}$$

Par conséquent, pour construire le système, cette méthode d'intégration commence par construire le vecteur `b`. Dans un premier temps, il lance la fonction `addForce` pour tous les composants qui héritent du composant `ForceField` pour calculer les forces internes. Dans un deuxième temps, il lance la fonction `addDForce` pour les mêmes composants pour calculer $kFacteur\frac{\partial F}{\partial U}$`v` avec donc en paramètre `kFacteur` comme facteur multiplicatif et `v` comme vecteur multiplicatif.
Une fois la vecteur`b` construit, cette méthode d'intégration est définie implicitement par la matrice `A` en indiquant les coefficients se trouvant devant les différentes matrices à sommer :

- le facteur multiplicatif de la matrice M est 1,
- le facteur multiplicatif de la matrice $\frac{\partial F}{\partial V}$ est 0, s'il n'y a pas d'amortissement,
- le facteur multiplicatif de la matrice $\frac{\partial F}{\partial U}$ est $-h^2$.

Une fois la matrice A construite, cette méthode d'intégration lance la méthode de résolution du système linéaire. Dans notre cas, il lance la méthode du gradient conjugué (voir l'annexe **??**). Cette méthode est une méthode itérative. Ainsi, le gradient conjugué lance la fonction `addDForce` autant de fois qu'il y a d'itérations. Les critères d'arrêt ont été vus dans la section précédente à la description du composant *CGLinearSolver*. Tout le temps de calculs de cette méthode est concentré sur le calcul de la multiplication entre la matrice des dérivées des forces et le vecteur multiplicatif V. C'est pour cela que nous avons parallélisé ce calcul.
La structure de `addForce` est très semblable à celle de `addDForce`. Le code 16 décrit l'algorithme principal de la fonction `addForce`. Cela est à mettre en parallèle avec le programme **??**, processus lancé sur le *GPU* expliqué dans le chapitre précédent.

Code 16 Structure de *addForce*.

```
1  //Pour chaque élément
2  for (numElement=0;numElement<getNbElements();numElement++)
3  {
4    //Récupère le numéro de l'association LED, les données et l'élément
5    numConteneurLED = getNumConteneurLED(numElement);
6    DataElement & d = getDataElement(numElement);
7    curElement = d.element;
8    //Déplacements d'un tétraèdre, hexaèdre, triangle, quadrangle
9    Coord u[curElement->nbNodes];
10
11   //Pour chaque noeud
12   for (n=0;n<curElement->nbNodes;n++)
13   {
14     //Pour chaque dimension
15     for (d=0;d<curElement->nbDim;d++)
16     {
17       //Position courante moins initiale
18       u[n][d] = (x[d.ind[n]][d]-d.P[n][d]);
19     }
20   }
21
22   //Pour chaque fragment
23   for (s=0;s<listEV[numConteneurLED].size();s++)
24   {
25     //Add the current contribution
26     listEV[numConteneurLED][s]->compute(f,u,d);
27   }
28 }
```

Des différences importantes existent entre le code *CPU* et le code *GPU*. Nous voulons en souligner deux. Premièrement, la ligne 2 sur le code *CPU* numéro 16 est une boucle pour chaque élément. Dans le code *GPU*, ceci un remplacé par le fait que le contexte *GPU* lance n processus, avec n le nombre d'éléments. Deuxièmement, la ligne 26 sur le code *CPU* numéro 16 accumule pour l'élément courant E, pour le fragment courant, l'influence

de E sur ses nœuds. Dans le code *GPU*, ceci un remplacé par le fait qu'il y a deux étapes : d'abord une étape de calcul des contributions partielles, ensuite une étape de sommation des contributions pour chaque nœud.

TrackRunSofa. Une fois la boucle de simulation lancée, l'utilisateur décide à quel moment il souhaite interrompre ou reprendre la simulation. Dans le cadre d'une batterie de tests, nous souhaitons que la simulation s'arrête lorsqu'elle est considérée comme stable. C'est le rôle du composant `TrackRunSofa`.
Ce composant permet d'analyser la stabilité de la simulation. Lorsque la simulation se termine, il écrit un fichier résultat qui contient les temps moyens de l'exécution des fonctions principales. Nous souhaitons avoir le temps moyen passé pour un pas de simulation, pour le calcul des forces internes et pour le calcul de la matrice des dérivées partielles. Ce sont des chronomètres qui sont placés au début et à la fin des fonctions qui permettent d'avoir ces informations.
Ce composant analyse en permanence l'énergie cinétique aux nœuds. Lorsque l'énergie moyenne est inférieure au paramètre `epsilonStable`, la simulation se termine. Avant de rendre la main au programme appelant, le composant `TrackRunSofa` écrit le fichier de résultats contenant les temps moyens d'exécution, lance une capture d'écran de la simulation pour avoir une visualisation de l'état du modèle et l'enregistrement des coordonnées finales du modèle pour chaque particule.
Pour suivre la simulation, de nombreux paramètres sont nécessaires. Dans un premier temps, la variable `periode` permet d'indiquer le temps entre deux analyses de la cinétique des particules, ce qui peut déclencher la fin de la simulation. La variable `epsilonStable` correspond à un seuil en-dessous duquel la simulation s'arrête si l'énergie cinétique moyenne pour chaque particule est inférieur à celui-ci. Les variables `nomMaillageDeforme` et `nomTemps` correspondent au nom des fichiers pour enregistrer les coordonnées des particules à la fin de la simulation et le temps moyen des chronomètres précédemment décrits. La dernière variable `tempsEtirement` est utile lorsque l'utilisateur défini une traction pour éviter que la simulation s'arrête avant la fin de la traction. Le code 17 `xml` est un exemple de l'intégration de tous ces paramètres dans une scène *SOFA*.

Code 17 Ajout du composant pour suivre la simulation.

```xml
<!-- Track the simulation -->
<TrackRunSofa listChrono="@variableTemps.listChrono" nomMaillageDeforme="maillageDeforme.msh"
    tempsEtirement="0.1" nomTemps="tpsEtirement.txt" periode="0.05" epsilonStable="0.005" />
```

Pour utiliser tous ces composants, il faut des maillages. Nous avons choisi de faire nos tests sur des barres hybrides mélangeant des lois de comportement et des éléments de base.

1.3 Génération de barres hybrides

Pour générer un maillage hybride, nous avons besoin de la géométrie, de la topologie et des informations sur les associations `LED` pour la loi de comportement, le type d'éléments et la manière de découper les équations générées. Nous avons décrit précédemment les trois formats de fichiers nécessaires à la définition d'un maillage hybride :

- `maillage.obj` : Ce fichier est utile pour le composant `MeshTopology` de *SOFA*. Il contient les nœuds du maillage.
- `maillage.element` : Ce fichier est lu par `MixTopology` et contient la topologie du maillage.
- `maillage.info` : Ce fichier est lu par `MixTopology` et contient les associations `LED` de chaque élément du maillage.

Pour utiliser les composants *TM* et *TMG* de *SOFA*, ces trois fichiers doivent exister. Les tests que nous allons réalisés sont effectués sur des barres. Ce sont uniquement des tests de validation de modèle. Nous avons construit un générateur de barres hybrides.

1.3.1 Entrées et sorties

Les trois fichiers précédemment décrits sont créés par notre générateur de barres hybrides. L'entrée du programme configure la barre à générer. Une barre hybride est en fait une juxaposition de barres non hybrides de même section composés d'éléments appartenant à la même association `LED`.
Une barre hybride est générée à partir des paramètres contenus dans un fichier de configuration. Le code 18 donne un exemple de fichier de configuration. Pour des raisons pratiques, ce fichier contient la variable `nomDossier` dans

lequel sont créés les fichiers et la variable `nomMaillage` qui donne la racine commune à chacun des fichiers générés pour ajouter de l'ordre.
La variable `longueur` permet de définir la longueur de la barre. La variable `"largeurSection"` permet de définir la largeur de la section de la barre. Nous générons des barres à profil carré. La variable `nbCubeLongueur` permet de définir le nombre total de cubes dans le sens de la longueur. La variable `nbCubeWithSection` permet de définir le nombre total de cubes dans le sens de la largeur ou de la profondeur.
Pour chaque sous barre non hybride SBH_i, nous définissons 4 paramètres. Le premier paramètre est le pourcentage de la longueur à partir duquel commence une nouvelle sous-barre. Si c'est la première barre, alors elle commence au début et se termine au pourcentage indiqué. Sinon, elle commence là où la sous-précédente s'est arêtée. La dernière sous-barre hybride a nécessairement le pourcentage en 1. Les trois paramètres suivants sont les noms des composants de l'association `LED` de la SBH_i : le composant `Loi` de comportement, le composant `Element` et le composant `Decoupage`.

Code 18 Exemple de fichier de configuration pour la génération de maillage.

```
nomDossier ./
nomMaillage barreHybride
longueur 6
largeurSection 1
nbCubeLongueur 12
nbCubeLargeurSection 2
0.5 Hooke Triangle DecoupageNon
1.0 SaintVenant Quad DecoupageNon
```

L'exemple du code 18 correspond à une barre 2D constituée de deux sous-barres non hybrides. La première est une barre dont la dimension est la moitié de celle de la barre. Elle est rattachée à l'association `LED`, $Hooke$, $Triangle$ et $DecoupageNon$. La seconde est une barre dont la dimension est la moitié de celle de la barre. Elle est rattachée à l'association `LED`, $SaintVenant$, $Quad$ et $DecoupageNon$ comme le montre la Figure 1.3 donnant une visualisation de la barre générée au sein de *SOFA*.

1.3.2 Algorithme pour la génération des barres hybrides

Le but du générateur est de générer les trois fichiers `maillage.obj`, `maillage.element` et `maillage.info` à partir du fichier de configuration explicitée précédemment. L'idée principale est de se baser sur une grille de nœuds. Certains sont des nœuds primaires d'autres sont des nœuds secondaires comme le montre la Figure 1.4(b). La grille des nœuds secondaires est deux fois plus fines que celle des nœuds primaires. La grille secondaire permet de construire les éléments qui possèdent des nœuds sur les arêtes. Cela correspond au nœuds d'interpolation. L'avantage de construire une grille est de pouvoir marquer les nœuds utilisés dans un premier et ensuite de pouvoir numéroter uniquement les nœuds qui ont été marqués.
Pour créer `maillage.element`, trois phases sont nécessaires :

- le code 19 permet d'ajouter dans un premier temps les éléments du maillage. Au début, la barre est définie comme étant une grille régulière de quadrilatères ou de cubes. Chacun d'eux va être divisés en fonction de l'association `LED` dans laquelle il se trouve. Le fichier de configuration définit le nombre du cube `nbCubelongueur` en x et `nbCubeLargeurSection` en y et en z. En 2D, le principe est le même, mais avec des quadrilatères. Pour un cube ou un quadrilatère, le programme :
  - récupère la position en x du barycentre de l'élément. Avec cette position, le programme connaît l'association `LED` située à cet endroit d'après les pourcentages indiqués dans le fichier de configuration des sous-barres non hybrides,
  - découpe le cube ou le quadrilatère en l'élément correspondant à l'association `LED`. Si l'élément est un triangle, alors le quadrilatère est découpé en deux triangles. Si l'élement est un quadrilaètre ou un hexaèdre, il est laissé inchangé. Si l'élément est un tétraèdre, alors le cube est découpé en 6 tétraèdres. Si l'élément est un prisme, alors le cube est découpé en deux prismes. Si l'élément est une pyramide, alors le cube est découpé en 6 pyramides. L'annexe **??** montre tous ces découpages.
  - calcule les indices de chaque nœud formant les éléments découpés précédemment en fonction de l'indice du quadrilatère ou du cube et de la grille nœuds.

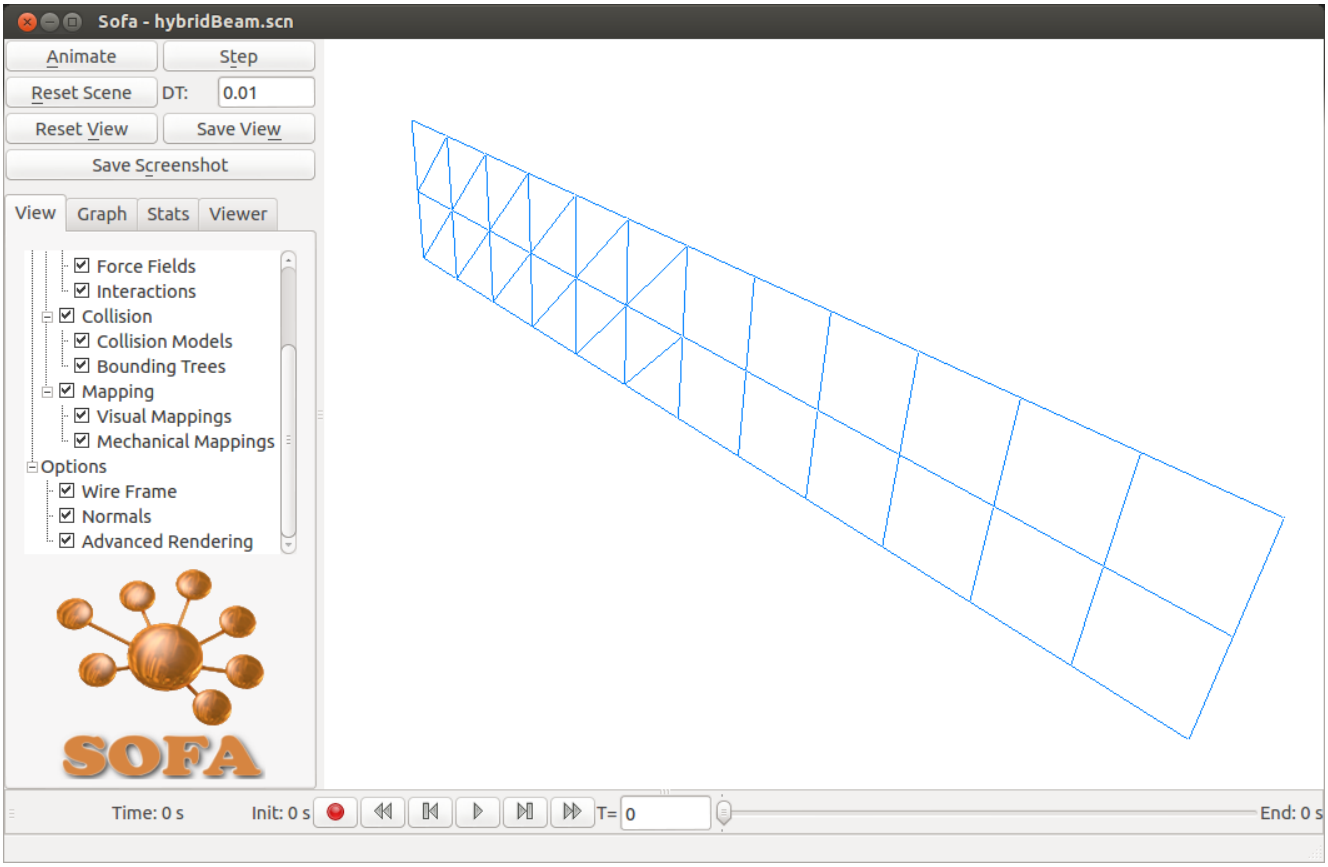
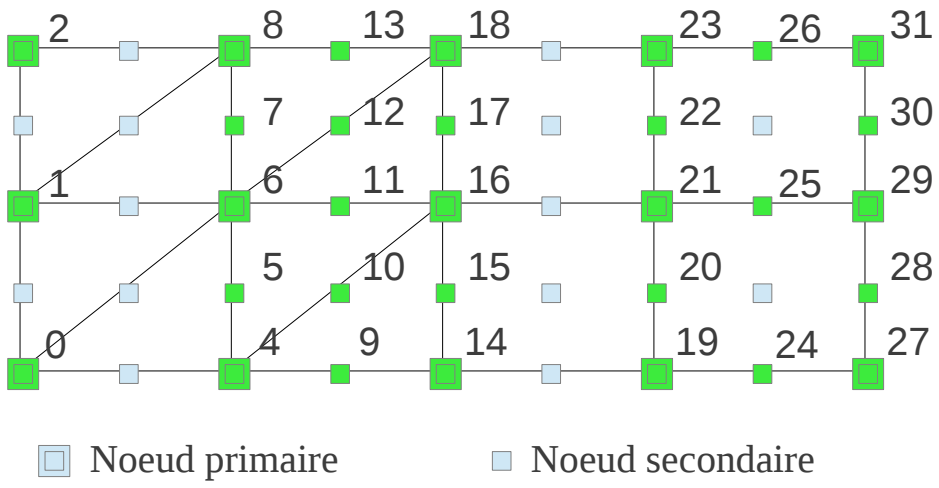
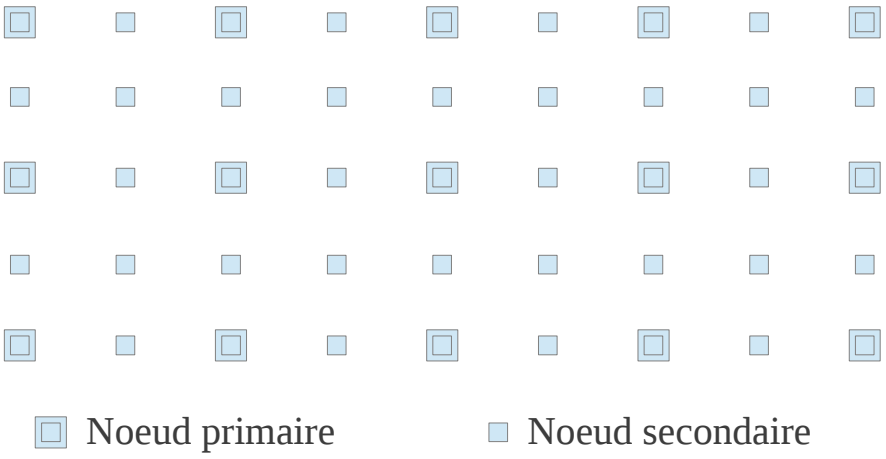


FIGURE 1.3 – Exemple d'une barre hybride en 2D avec *SOFA*.



(a) Exemple de barre basée sur une grille de nœuds.



(b) Grille de nœuds permettant de générer la barre Figure 1.4(a).

FIGURE 1.4 – Construction d'un maillage 2D avec une grille primaire et une grille secondaire

- le code 20 marque les nœuds utilisés dans le but de numéroter chacun des données pour que chaque élément puisse connaître l'indice final de ces nœuds. En effet, dans la grille de nœuds, certains ne seront pas ajoutés dans le fichier définissant l'ensemble des particules du maillage, ce qui va changer l'indexation des particules. Grâce à l'ajout des éléments, nous savons quels sont les nœuds qui sont utilisés et ceux qui ne le sont pas. Pour chaque élément, pour chaque nœud, nous marquons le nœud primaire ou secondaire comme étant utilisé.
- le code 21 numérote les nœuds lorsqu'ils sont utilisés. Chaque nœud possède un attribut `used`. Si cet attribut est à vrai, alors la phase précédente l'a identifié comme appartenant à un élément du maillage. Pour chaque colonne en x, pour chaque colonne en y et pour chaque profondeur en z, nous récupérons le nœud. S'il est utilisé, alors nous lui donnons le numéro correspondant au nombre de nœuds utilisés déjà parcourus comme le montre la Figure 1.4(a).

Code 19 Ajout des éléments en fonction de la sous-barre.

```
//Pour chaque cube en x
for (numCube[0] = 0; numCube[0] < nbCube[0] ; numCube[0]++ )
{
  //Récupère la sous-barre du cube courant
  meshZone = getZone(numCube[0]);

  //Pour chaque cube en y
  for (numCube[1] = 0; numCube[1] < nbCube[1] ; numCube[1]++ )
  {
    //Pour chaque cube en z
    for (numCube[2] = 0; numCube[2] < nbCube[2] ; numCube[2]++)
    {
      //Ajoute l'élément courant
      meshZone->addElement(numCube,vectorIndexElement);
    }
  }
}
```

Code 20 Marquage des nœuds utilisés.

```
//Pour chaque élément
for (numElement=0; numElement < vectorIndexElement.size(); numElement++ )
{
  //Pour chaque noeud
  for (numIndex=0;numIndex<vectorIndexElement[numElement].index.size();numIndex++)
  {
    //On récupère les indices en x y et z du noeud courant
    IndexVertex & i = vectorIndexElement[numElement].index[numIndex];
    //On le note comme étant utilisé
    baseVerticesUsed[i.index[0]][i.index[1]][i.index[2]] = true;
  }
}
```

Une fois les barres générées, nous pouvons générer des groupes de simulations utilisant des barres hybrides et appliquant des scénarios de base.

1.4 Génération, simulation d'un groupe de simulations

Un des objectifs de cette thèse est de tester rapidement un ensemble de méthode de résolution en fonction de la loi de comportement, du type d'éléments et du matériel utilisé. Le benchmarking doit comparer les temps de calculs

Code 21 Numérotation des nœuds.

```
// Initialisation
verticesUsed = 0;
//Pour chaque noeud en x
for (numVertex[0] = 0; numVertex[0] < nbPointBase[0] ; numVertex[0]++ )
{
  //Pour chaque noeud en y
  for (numVertex[1] = 0; numVertex[1] < nbPointBase[1] ; numVertex[1]++ )
  {
    //Pour chaque noeud en z
    for (numVertex[2] = 0; numVertex[2] < nbPointBase[2] ; numVertex[2]++)
    {
      //Si le noeud est utilisé
      if (baseVerticesUsed[numVertex[0]][numVertex[1]][numVertex[2]])
      {
        //On numérote le noeud courant
        index[numVertex[0]][numVertex[1]][numVertex[2]] = verticesUsed;
        //On ajoute un noeud en plus dans le maillage
        verticesUsed++;
      }
    }
  }
}
```

dans différentes zones du code et la précision. Pour comparer la précision, nous utilisons le test de traction. Le module de Young et le coefficient de Poisson peuvent être recalculés lors d'un test de traction. Cela donne une indication sur la précision de la méthode. De la même manière, la flèche lors d'un test de gravité permet de vérifier la cohérence des lois de comportement en fonction de la résolution. Tous les tests sont effectués sur des barres.
Le programme de génération de tests et de simulations est appelé `TestSimulation`. Il gère des entrées et des sorties, s'interface avec une base de données et lance les simulation par l'intermédiaire d'un exécutable de *SOFA*, $runSofa$.

1.4.1 Entrées et sorties

L'objectif est de :

- générer un ensemble de simulations,
- exécuter les simulations,
- récupérer les résultats pour chaque simulation.

Entrées. Avant toute chose, le programme a besoin de connaître le dossier dans lequel il va travailler ainsi d'une description du groupe de simulations que l'utilisateur veut lancer. Ces premières entrées sont données à l'exécution de `TestSimulation` par les arguments. La syntaxe est `./TestSimulation nomDossier description`. La variable `nomDossier` permet d'organiser les résultats dans des dossiers. L'utilisateur donne la description de son groupe de simulations dans la variable `description`.
Une simulation est paramétrée par un ensemble `p` de paramètres. Notons `nbp` le nombre de paramètres `p`. Un groupe de simulations est une suite de simulations qui prennent des valeurs de paramètres différentes. Pour définir l'ensemble des paramètres, l'utilisateur définit dans un premier temps la borne inférieure, la borne supérieure et le pas de chacun des paramètres `p`. Cela définit un ensemble `P` qui contient différentes configurations de l'ensemble des paramètres `p`. Notons le nombre de simulations `nbP`. Un groupe de simulations contient `nbP` simulation. Pour chaque simulation, plusieurs barres sont testées.
L'ensemble P est générer à partir des fichiers :

- Le tableau 1.1 définit le contenu du fichier `params.csv`. Ce fichier contient les paramètres qui définissent la simulation de la barre.
- Le code 22 définit le contenu du fichier `barres.csv`. Ce fichier contient la liste des barres à simuler.

- Pour chaque barre, il y a un fichier `Barre/nomDeLaBarre.csv`. Le code 23 décrit le contenu de ce fichier qui contient les informations de la barre à simuler. Le contenu ressemble au fichier 18 de configuration de la génération d'un barre.

nom paramètre	type	début	fin	pas	ordre	opération
nbCube	0	40	200	2	100	0
longueur	1	1.2	0	0.5	300	1
largeurSection	1	0.2	0	0.05	400	1
densite	1	800	0	0	200	1
moduleYoung	1	1000000	3000000	1000000	50	1
PoissonRatio	1	0.20	0	0.1	600	1
tailleLocale	0	64	0	2	700	0
pourcentage	1	0.05	0	0.2	850	1
tempsEtirement	1	0.9	0	1	852	1
epsilonStable	1	0.06	0	1	853	1
pas	1	0.01	0	2	854	0
typeSimulation	1	1	0	1	900	1
typeIntegration	0	1	0	1	950	1

TABLE 1.1 – Fichier `params.csv`.

Code 22 Fichier contenant la liste des barres à simuler.

```
#Ajouter votre barre ici
BarreSimple2D
HookeSVT3D
```

Code 23 Fichier exemple du contenu de `BarreSimple2D.csv`.

```
typePU GPU
component TM
category TM
1.0 Hooke Triangle DecoupageNon
```

Le fichier `params.csv` contient les variables qui définissent les paramètres du groupe de simulations correspondant. Il y a un premier ensemble de paramètres qui décrivent la géométrie globale de la barre. Nous avons déjà présenté 2 de ces variables. Il s'agit des variables `longueur` et `largeurSection`. À ces 2 variables, s'ajoute les propriétés du matériau de la barre définies par la variable `moduleYoung` et la variable `PoissonRatio`. Le fichier décrit également la densité volumique du matériau par la variable `densite` et le nombre total de cubes pour cette barre par la variable `nbCube`.
Il y a un deuxième ensemble de paramètres qui décrivent la comportement général de la simulation. Nous avons déjà vu la variable `epsilonStable` qui définit un seuil concernant l'arrêt de la simulation considérée comme étant stable. Il y a la variable `pas` qui correspond à la durée d'un pas de temps de simulation. Enfin, la variable `typeIntegration` vaut 0 si la méthode d'intégration souhaitée est Euler explicite. Pour toute autre valeur, la méthode d'intégration Euler implicité est utilisée.
Il y a un troisième ensemble de paramètres qui décrivent le type de simulation à réaliser et les caractéristiques propres à certains types de simulation. La variable `typeSimulation` est un entier qui définit le scénario de test à effectuer. L'entier 0 définit le test de traction, l'entier 1 définit le test de gravité, $\cdots$ Pour le test de traction, deux variables supplémentaires sont définies. La variable `pourcentage` est le pourcentage d'étirement par rapport à la `longueur` et la variable `tempsEtirement` est le temps durant lequel elle est étirée.
L'idée générale est que pour chaque paramètre, l'utilisateur définisse une zone à explorer pour tirer des conclusions sur l'influence de tel ou tel paramètre sur la simulation. Chacun de ces paramètres est définit comme le montre le tableau 1.2 grâce à plusieurs attributs. Un attribut essentiel et le `nom` du paramètre. La variable `type` permet de

savoir si le paramètre est un entier ou un réel. Si le paramètre est un entier, alors la variable `type` vaut 0 sinon elle vaut 1.
Deux variables servent à délimiter la zone d'exploration du paramètre. La variable `début` est la valeur initiale que prend le paramètre. La variable `fin` est la valeur finale du paramètre. Pour définir comment explorer les valeurs entre `début` et `fin`, nous avons deux autres variables. La variable `pas` définit l'intervalle entre chaque valeur. La variable `opération` définit l'intervalle comme étant une somme ou un produit. Si la valeur de la variable `opération` est égale à 0, alors la nouvelle valeur du paramètre est égale à l'ancienne plus le pas. Sinon, la nouvelle valeur du paramètre est égale à l'ancienne multiplié par le pas. Une dernière variable sert à classer les paramètres par ordre. Cela joue un rôle lors de l'exploration d'un ensemble de paramètres.
Prenons la première ligne du tableau 1.2. Le paramètre qui a l'`ordre` le plus petit est `nbCube`. C'est un entier vu que son `type` est égale à 0. Sa valeur initiale est 40. Lorsque la valeur dépasse 200, la valeur suivante retourne à la valeur initiale, c'est-à-dire 40 dans notre exemple. Le `pas` est de 2. La valeur de l'`ordre` est 100. L'`opération` est la multiplication puisque la valeur entrée est 0. La liste des valeurs prises par ce paramètre est 40, 80, 160, 320 puis de nouveau 40 et ainsi de suite jusqu'à ce que tous les paramètres aient été explorés. Dans l'exemple du tableau, nous avons choisi de faire varier uniquement 2 paramètres, `nbCube` et `E`. En effet, tous les autres paramètres sont déjà à la fin de l'exploration. La ligne correspondante au paramètre `E` indique que la valeur initiale est 1000000 avec un `pas` de 1000000 mais l'`opération` addition. La liste des valeurs est 1000000, 2000000 et 3000000. Ainsi, le nombre de cas exploré est $4 \times 3 = 12$, soit le tableau suivant :

Numéro	0	1	2	3	4	5	6	7	8	9	10	11
`nbCube`	40	80	160	320	40	80	160	320	40	80	160	320
`longueur`	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
`largeurSection`	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
`densite`	800	800	800	800	800	800	800	800	800	800	800	800
`moduleYoung` (MPa)	1	1	1	1	2	2	2	2	3	3	3	3
`PoissonRatio`	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
`tailleLocale`	64	64	64	64	64	64	64	64	64	64	64	64
`pourcentage`	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
`tempsEtirement`	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
`epsilonStable`	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
`pas`	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
`typeSimulation`	1	1	1	1	1	1	1	1	1	1	1	1
`typeIntegration`	1	1	1	1	1	1	1	1	1	1	1	1

TABLE 1.2 – Fichier `params.csv`.

Le fichier 22 définit deux barres. Le nombre total de simulations est donc de 24. Pour chaque simulation, des variables secondaires découlent des paramètres primaires p. Par exemple, à partir de `longueur` et de `largeurSection`, nous calculons le volume. Avec la densité volumique, nous calculons enfin la masse qui est un paramètre secondaire. Le code 24 donne un exemple de graphe de scène *SOFA* utilisant les variables primaires et secondaires.
Les paramètres primaires nécessaires pour la définition du composant `TM` ou `TMG` scène sont les variables `moduleYoung` et `PoissonRatio`, les propriétés du matériau. La variable primaire `tailleLocale` est utile pour le composant `TMG`, implémentation *GPU* de notre méthode. La variable primaire `pas` est directement utilisée dans le nœud racine de la scène *SOFA*. Les variables primaires `tempsEtirement`, `epsilonStable`, `periode` sont utilisés dans le composant `TrackRunSofa`.
La première variable secondaire est la variable `gravite`. Elle vaut 0 dans la cas d'un test de traction lorsque `typeSimulation` vaut 0. Sinon elle est à égale à la constante gravitationnelle de la Terre, soit $-9.81 m.s^{-2}$. La variable secondaire `nomDossier` est la concaténation de la variable `nomDossier` suivi de la chaîne `"Result/"` suivi du nom de la simulation courante et suivi du caractère `"/"`. C'est le nom du dossier de la simulation courante. La variable secondaire `nomMaillage` est le nom de la barre courante simulée. Ce nom est contenu dans le fichier `barres.csv`.
La variable secondaire `masse` est la masse de la barre calculée à partir des valeurs courantes de la `longueur`, de la `largeurSection` et de la `densite`. La masse est fonction égale de la dimension de la barre. Si c'est une barre 2D, nous calculons d'abord l'aire, sinon nous calculons le volume.
Pour pouvoir fixer les points de la face gauche de la barre ou tirer les points de la face droite de la barre, nous avons besoin des indices de ces points. La variable secondaire `indicesGauches` contient la liste des indices de la face gauche. Les points de cette face ont leur abscisse négative. La barre est centrée par rapport à l'origine. La variable secondaire `indicesDroits` contient la liste des indices de la face droite. Les points de cette face ont leur abscisse positive.

Le dernier paramètre secondaire est nécessaire uniquement pour le test de traction. La variable `deplacement` est le produit entre les valeurs courantes de `longueur` et `pourcentage`. Cette variable sert à connaître le déplacement en x des points de la face droite.
À chaque simulation, le processus suivant s'opère :

- Le dossier correspondant à la simulation courante est créé pour contenir toutes ses entrées et toutes ses sorties.
- À partir du nom du dossier `nomChemin` et des paramètres de la géométrie et de la loi de comportement, les fichiers liés au maillage sont générés.
- À partir des paramètres secondaires, le programme génère le squelette de scène 24 `simulation.scn` de *SOFA*.
- Une fois le fichier scène généré, la commande `runSofa simulation.scn > synchro.txt` est lancée.

Sorties. Pour chaque simulation de nombreuses sorties sont générées. Quand le programme de *SOFA* `runSofa` se termine, trois fichiers de résultats sont créés :

- Lorsqu'il considère la simulation stable, `TrackRunSofa` enregistre le fichier scène à l'état courant, c'est-à-dire avec les positions courantes des nœuds du maillage. Le nom de ce fichier est l'attribut `sceneFileName` de ce composant.
- Ensuite, pour chaque chronomètre, il enregistre le temps d'exécution moyen précédé du nom du chronomètre. Dans la sous-section 1.2.3, nous rappelons l'utilité des 3 chronomètres `stepTime`, `addForce` et `addDForce`. Le nom du fichier créé est l'attribut `timeFileName` de ce composant.
- Enfin, `TrackRunSofa` fait une capture d'écran de la fenêtre de `runSofa` comme le montre la Figure 1.5. Par la suite, il lance la fin du programme `runSofa` pour donner la main au programme appelant `TestSimulation`.

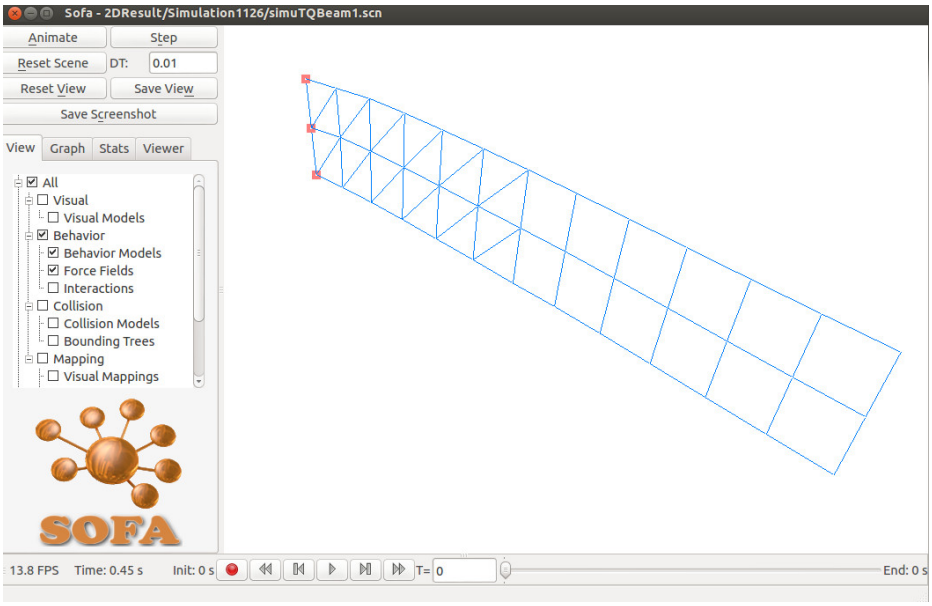


FIGURE 1.5 – Exemple maillage 2D.

À partir du fichier `sceneFileName`, les positions pour chaque nœud sont récupérées. Elles servent pour deux occasions :

- Dans un premier temps, nous enregistrons les positions de la flèche pour le test de gravité. Pour chaque section, nous récupérons la position moyenne comme le montre la Figure 1.6.
- Dans un second temps, nous calculons le module Young lors d'un test de traction. À partir des indices des faces droite et gauche, nous récupérons les forces appliquées sur ces faces. Avec la formule de l'élongation (1.2), nous pouvons calculer le module de Young pour chaque simulation. Il s'agit par la suite de comparer avec les données réelles pour juger de la précision de la simulation. F est la force totale exercée sur une des deux faces. A_0 est la surface de section à l'origine avant l'application de la force. δL est l'élongation de la barre. L_0 est la longeur de la barre à l'origine.

Code 24 Squelette de la scène définissant la simulation courante.

```
<?xml version="1.0" ?>
<!-- Voir http://wiki.sofa-framework.org/mediawiki/index.php-->
<Node name="root" gravity="0 $gravity 0" dt="$step" animate="1" >
  <VisualStyle displayFlags="hideVisualModels showBehaviorModels showForceFields" />

  <!-- Variable qui contient les temps moyens d'exécution -->
  <VariableTimer name="variableTimer" simunomDossier="$nomChemin" methodName="$nomMaillage" />

  <!-- Méthode d'intégration -->
  <EISolver listChrono="@variableTimer.listChrono"
    name="solver" printLog="false" vdamping="3" />
  <CGLinearSolver template="GraphScattered" name="linear solver"
    iterations="100" tolerance="1e-6" threshold="1e-6"/>

  <!-- Géométrie -->
  <MeshTopology name="mesh" fileTopology="$nomMaillage + Node.obj"/>
  <!-- Vecteur d'état -->
  <MechanicalObject name="dofs"/>
  <!-- Topologie -->
  <MixTopology name="MixTopology" meshName="$nomMaillage"/>

  <!-- Calcul des forces internes sur GPU -->
  <TMG name="TMG" listChrono="@variableTimer.listChrono"
    E="$moduleYoung" PoissonRatio="$PoissonRatio"
    localSize="$tailleLocale" />

  <!-- Mass component -->
  <UniformMass name="mass" totalmass="$masse"/>

  <!-- Composant qui étudie la stabilité de la simulation -->
  <TrackRunSofa listChrono="@variableTimer.listChrono"
    sceneFileName="res+$nomMaillage+.scn"
    timeStretch="$tempsEtirement" timeFileName="tm+$nomMaillage+.txt"
    period="$periode" epsilonStable="$epsilonStable" />

  <!-- Dans le cas d'une traction -->
    <PartialFixedConstraint fixedDirections="1 0" indices="$indicesGauches" />
    <PartialLinearMovementConstraint movedDirections="1 0 "
        printLog="false" indices="$indicesDroits" >
      <Attribute type="keyTimes">
        <Data value="0 $timeStretch 1000000000000" />
      </Attribute>
      <Attribute type="movements">
        <Data value="0 0  $deplacement 0  $deplacement 0"/>
      </Attribute>
    </PartialLinearMovementConstraint>
  <!-- else -->
    <FixedConstraint indices="$indicesGauches" />
  <!-- Fin du cas d'une traction -->
</Node>
```

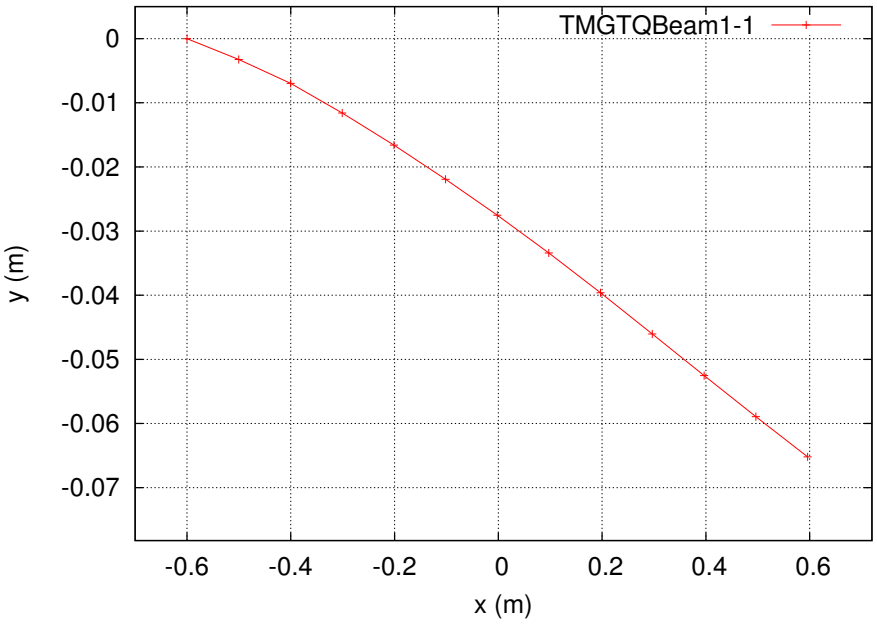


FIGURE 1.6 – Exemple maillage 2D.

$$E_{calc} \equiv \frac{\text{Stress}}{\text{Contrainte}} = \frac{\sigma}{\varepsilon} = \frac{F/A_0}{\Delta L/L_0} = \frac{FL_0}{A_0 \Delta L} \tag{1.2}$$

L'Algorithme 1 montre comment passer d'un jeu de paramètres à un autre, une fois que toutes les sorties sont enregistrées. Deux fonctions sont essentielles au fonctionnement de l'algorithme :

- `currentParam = nextParam(currentOrder)` : cette fonction permet de récupérer le paramètre suivant en fonction de l'ordre courant. L'initialisation ce fait à 0. L'ordre d'un paramètre doit être strictement positif lorsqu'il est renseigné dans le fichier `params.csv`.
- `nextValParam(currentParam)` : cette fonction incrémente la valeur courante du paramètre. Si l'`opération` du paramètre est une multiplication alors la valeur courante est multipliée par son `pas`. Sinon la valeur courante est ajoutée à son `pas`.

Algorithme 1 Algorithme pour aller d'un jeu de paramètres à un autre.

```
%Initialisation
currentOrder = 0 ;
currentParam = nextParam(currentOrder) ;
incrementation = false ;
Tant que pas d'incrementation et il y a des paramètres Faire
  Si valeur du paramètre dépasse le maximum Alors
    currentParam.val = currentParam.minVal ;
  Sinon
    nextValParam(currentParam) ;
    incrementation = true ;
  currentParam = nextParam(currentOrder) ;
Si pas d'incrementation Alors
  %Le programme a généré l'ensemble des jeux de paramètres possibles
  endLoop = true ;
```

À ce stade, nous sommes capables de générer un groupe de simulations. Les résultats des simulations ainsi que les paramètres qui ont permis d'effectuer les simulations sont stockés dans une base de données.

1.4.2 Base de données

La modularité est le mot-clé de notre plateforme. Le groupe de simulations a été paramétré au maximum pour permettre l'exécution de simulations avec un scénario donné pour effectuer les principaux tests de la mécanique (traction, flexion, torsion, ...). Pour organiser toutes les données en entrée et en sortie, la Figure 1.7 montre le schéma relationnel de notre base de données.

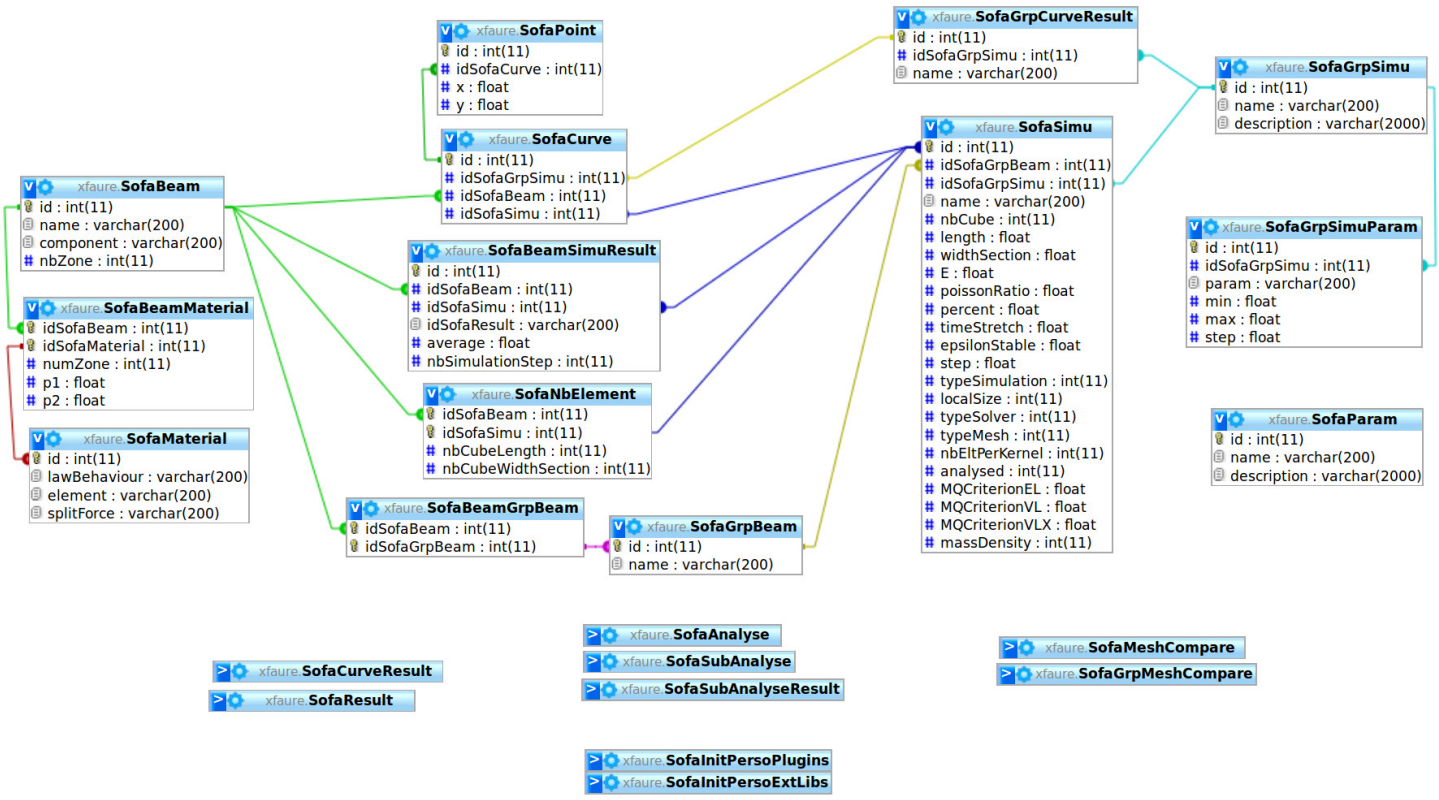


FIGURE 1.7 – Schéma simplifié de la base de données.

La table principale est `SofaSimu`. Pour chaque jeu de paramètres, un nouvel élément est inséré. Cet élément est lié à un groupe de simulations par la table `SofaGrpSimu` et à un groupe de barres par la table `SofaBeamGrpBeam`. Chaque barre est stockée dans la table `SofaBeam`. Une barre est liée à plusieurs associations `LED`. Une association `LED` est stockée dans la table `SofaMaterial`. À partir d'une barre et d'un jeu de paramètres, nous pouvons stocker le nombre de cubes ou quadrilatères pour la longueur et pour la dimension du carré de la section. Ces valeurs sont stockées dans la table `SofaNbElement`. Pour chaque groupe de simulations, le fichier `params.csv` est stocké dans la table `SofaGrpSimuParam`. Elle contient pour chaque paramètre la valeur de début, la valeur de fin et le pas.
Les sorties les plus simples à stocker sont les temps moyens dans les fonctions `addForce`, `addDForce` et `stepTime`. Ces valeurs sont stockées dans la table `SofaBeamSimuResult`. Pour stocker la flèche de la barre, nous utilisons une table `SofaCurve` liée à `SofaBeam` et `SofaSimu`. Pour chaque point de la flèche, un point est inséré dans la table `SofaPoint`, liée à la table `SofaCurve`. D'autres tables servent pour le stockage des analyses. Nous présentons ces tables dans la section suivante.

1.5 Analyse d'un ensemble de simulations

Grâce à notre base données, nous pouvons analyser les données de manière organisé. Dans un premier temps, à l'aide d'un logiciel, nous sélectionnons les entrées et les résultats à générer. Dans un deuxième temps, le programme réalise l'opération et génère des graphes de comparaison de flèches, de temps et de paramètres physiques calculés.

1.5.1 Entrées

La Figure 1.8 montre la fenêtre principale qui permet de sélectionner le groupe de simulations à analyser à l'aide d'une liste liée à la base de données. Une fois le groupe de simulations sélectionné, 4 zones de l'interface sont

actualisées :

- La zone la plus haute permet d'afficher les paramètres principaux. Ce sont les paramètres qui varient. Pour savoir si un paramètre varié, nous regardons sa valeur de `début` et sa valeur de `fin`. Si ces deux valeurs sont différentes, alors c'est un paramètre qui varie. Les valeurs sont stockées dans la table `SofaGrpSimuParam`.
- La zone description permet de charger l'argument tapé par l'utilisateur lors de l'exécution de `TestSimulation`. Cet argument décrit de manière globale le but de ce groupe de simulations.
- Au mileu de la fenêtre, la liste `Beam` contient les barres présentes dans le groupe de simulations sélectionné.
- À droite de la liste `Beam`, la liste `Parameter` contient la liste des paramètres liée à la simulation.

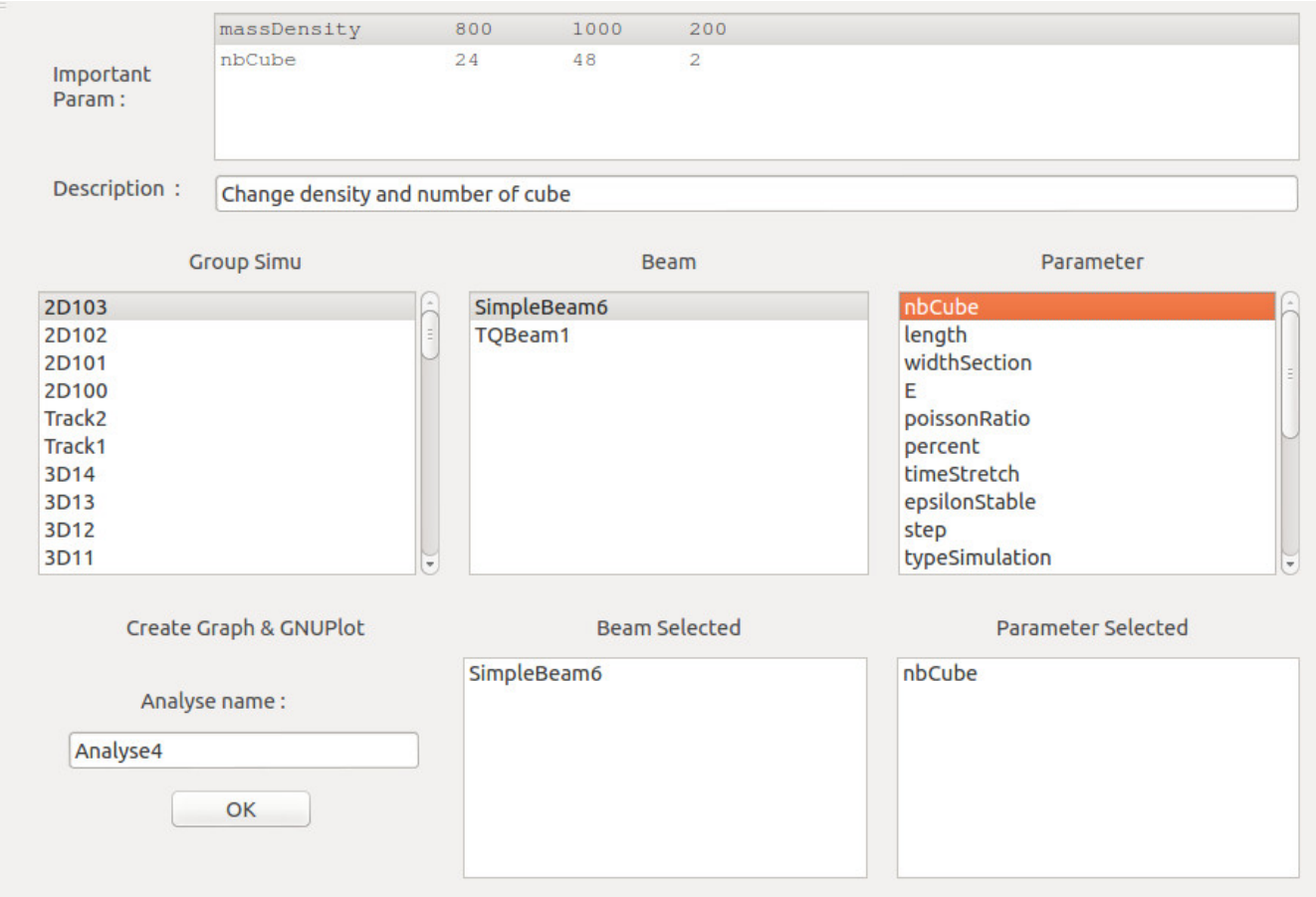


FIGURE 1.8 – Interface du programme d'analyse de données.

Les deux premières zones sont descriptives. Ces zones ne peuvent pas être modifiées. Les deux dernières zones permettent de sélectionnées les barres à analyser et les axes sur lesquelles elles vont être analysées. Pour sélectionner une barre, il faut le glisser de la liste `Beam` à la liste `Beam Selected`. Il en est de même pour sélectionner un paramètre. Une fois que les entrées sont prêtes, l'utilisateur appuie sur le bouton `"OK"` pour lancer le processus d'analyse.

1.5.2 Sorties

Les sorties sont des graphes. Il y a trois types de graphe :

- la Figure 1.9 présente les graphes de comparaison de paramètres de temps ou des paramètres physiques calculés.
- la Figure 1.10 illustre les graphes de flexion sur lesquels nous pouvons faire une comparaison qualitative et visuelle.
- la Figure 1.11 montre les graphes d'erreur pour visualiser l'influence de la résolution sur la précision.

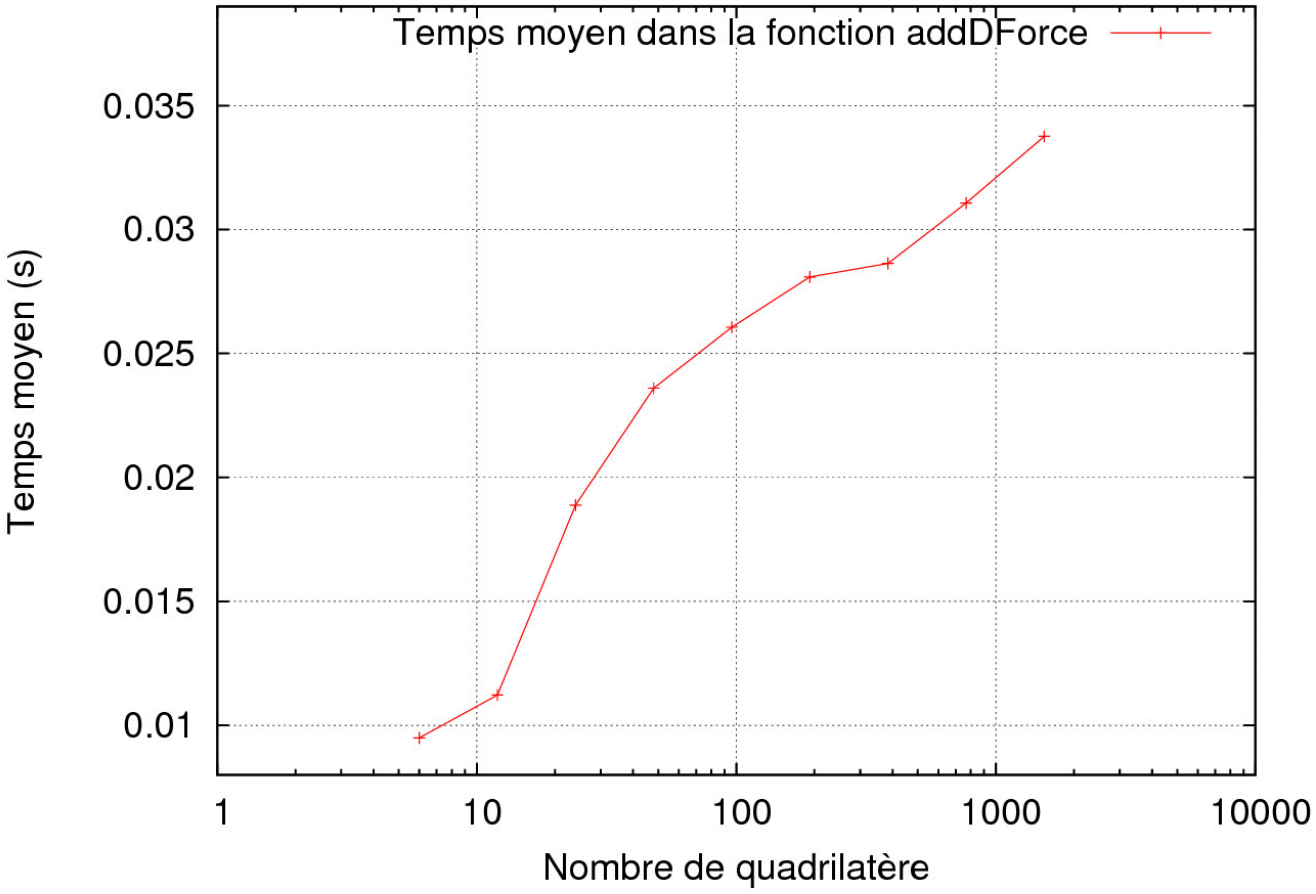


FIGURE 1.9 – Graphe du temps moyen en fonction du nombre de quadrilatère.

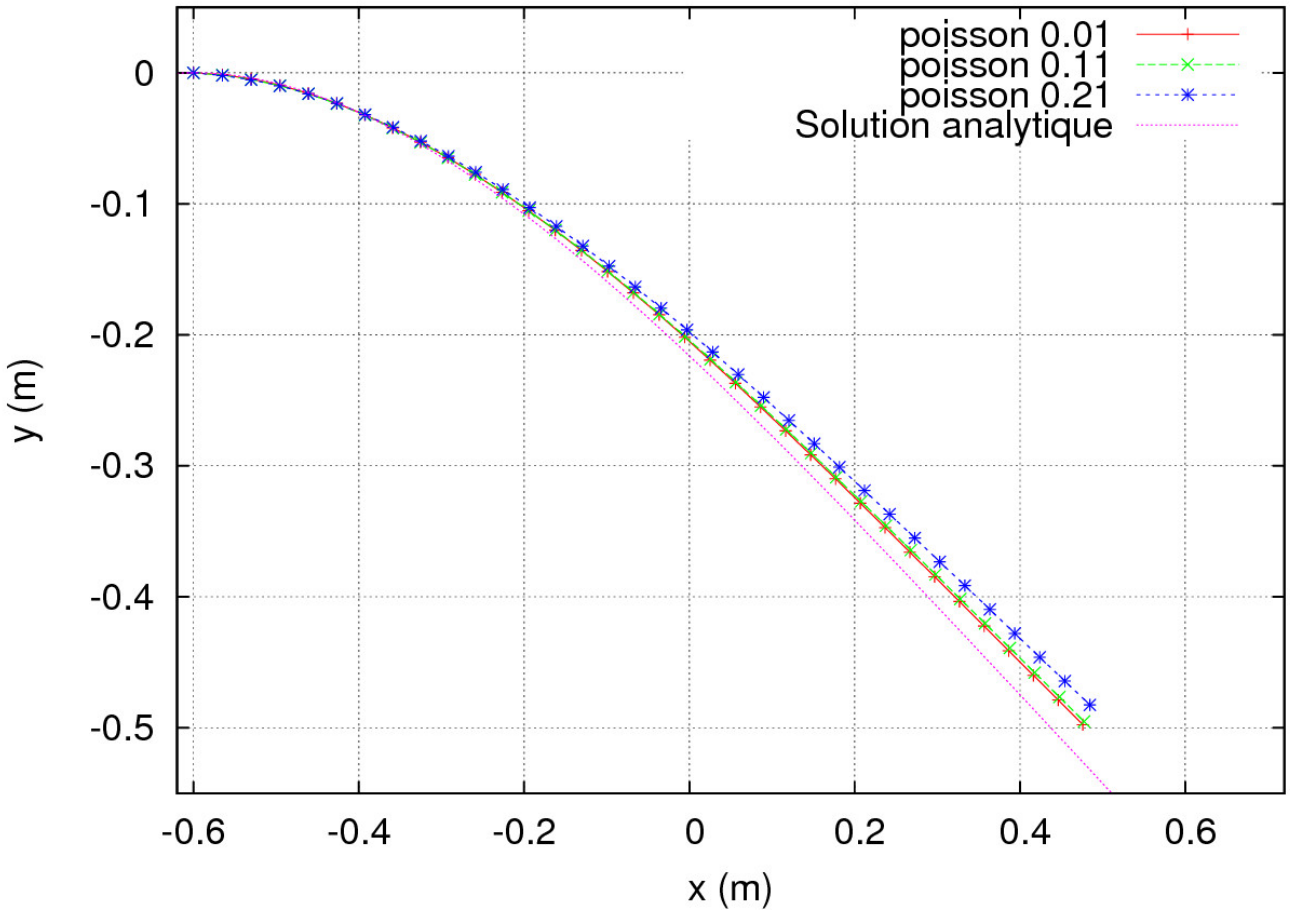


FIGURE 1.10 – Flexion d'une barre selon le coefficient de Poisson.

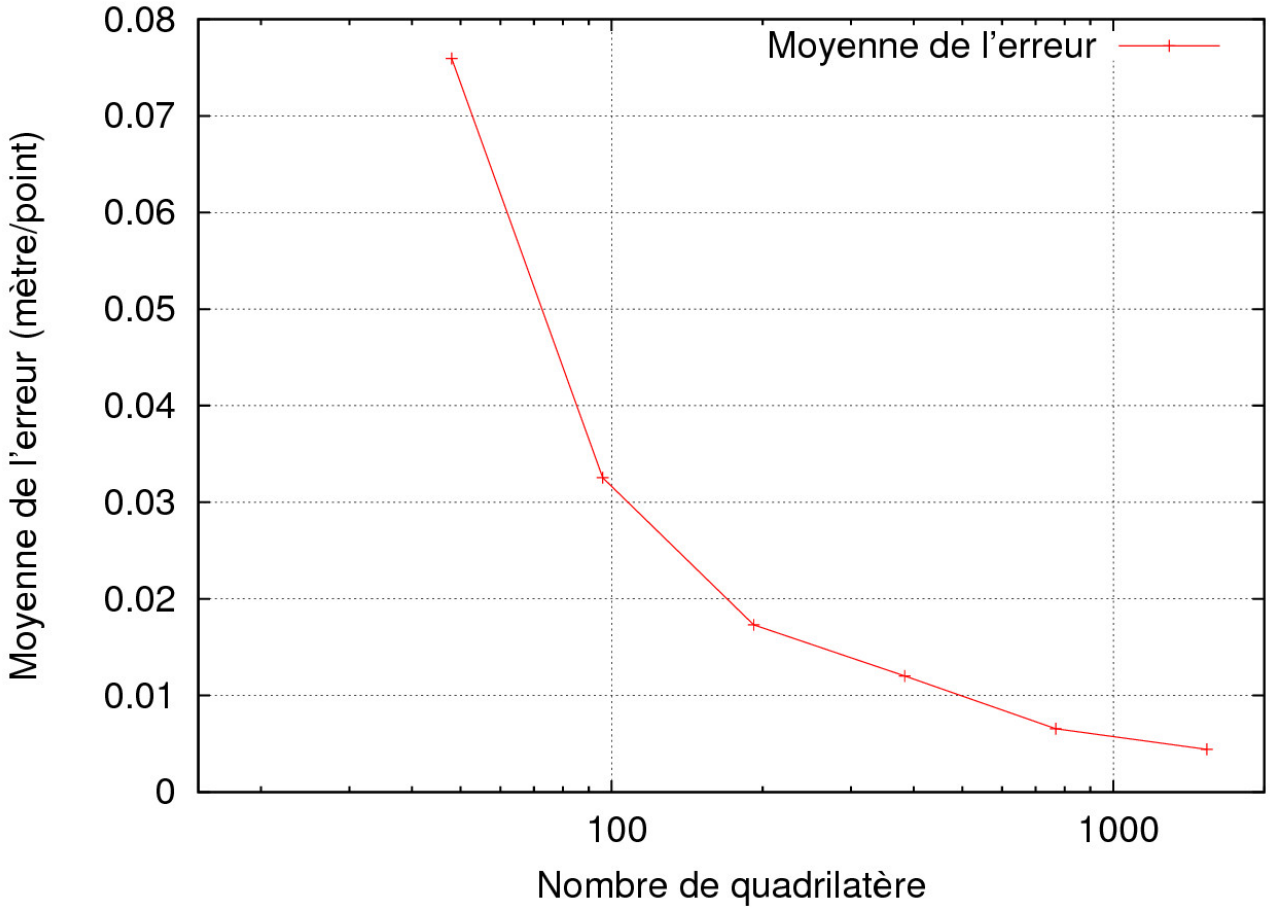


FIGURE 1.11 – Écart moyen entre une barre et la barre la plus résolue du groupe de simulations.

1.5.3 Génération de graphes résultats

L'utilisateur clique sur le bouton `"OK"` pour lancer la génération les graphes de résultat en fonction de paramètres d'entrées. La première étape consiste à insérer dans la base de données une nouvelle analyse. L'insertion d'un tuple dans la table `SofaAnalyse` génère un identifiant `idAnalyse` et un nom unique `nameAnalyse`. Ce nom unique est le nom du dossier à créer pour contenir les fichiers de l'analyse.
À partir de l'identifiant du groupe de simulations, nous pouvons déterminer :

- le paramètre le plus variant, celui qui possède le plus grand nombre de valeur discrète. Ce paramètre est le paramètre principal, `mainParam`. Il est déterminé en fonction de sa valeur `début`, de sa valeur `fin`, de son `pas` et de son `opération`.
- les paramètres variants, ceux qui possèdent au moins deux valeurs discrètes, excepté `mainParam`. Ces paramètres sont les paramètres variables, `variableParams`.
- les paramètres fixes, ceux qui n'ont qu'une seule valeur. Ces paramètres sont les paramètres fixes, `fixedParams`.

Dans l'exemple 1.1, `nbCube` est le `mainParam`. Ses valeurs discrètes sont 40, 80, 160 et 320, soit 4 valeurs. Il n'y a qu'une variable dans `variableParams`, `E`. Ses valeurs discrètes sont 1000000, 2000000 et 3000000, soit 3 valeurs. Tous les autres paramètres sont dans $fixedParams$.
Grâce à l'utilisation de la base de données, nous créons une vue particulière `SofaViewFixedSimus`. Cette vue est la jointure entre `SofaSimu` et `SofaBeam` suivi de `mainParam` et de `variableParams`. Dans le cas de l'exemple 1.1, cela donne les 12 simulations comme nous l'avions expliqué dans la sous-sous-section 1.4.1 illustrée par la Figure 1.12.

←T→	idSofaBeam	idSofaSimu	E	nbCube
Modifier Éditer en place Copier Effacer	8	387	1000000	40
Modifier Éditer en place Copier Effacer	8	388	1000000	80
Modifier Éditer en place Copier Effacer	8	389	1000000	160
Modifier Éditer en place Copier Effacer	8	390	1000000	320
Modifier Éditer en place Copier Effacer	8	391	2000000	40
Modifier Éditer en place Copier Effacer	8	392	2000000	80
Modifier Éditer en place Copier Effacer	8	393	2000000	160
Modifier Éditer en place Copier Effacer	8	394	2000000	320
Modifier Éditer en place Copier Effacer	8	395	3000000	40
Modifier Éditer en place Copier Effacer	8	396	3000000	80
Modifier Éditer en place Copier Effacer	8	397	3000000	160
Modifier Éditer en place Copier Effacer	8	398	3000000	320

FIGURE 1.12 – Exemple de vue `SofaViewFixedSimus`.

Pour chaque combinaison possible avec les valeurs des `variableParams`, nous créons autant de graphe qu'il y a de résultats différents dans la table `SofaResult` avec autant de barres qu'il y en a dans le groupe de simulations, ce qui donne la Figure 1.13.
Prenons le résultat `E`, le module d'Young recalculé grâce à la formule (1.2). Ce résultat n'a du sens que pour un test de traction. Pour notre exemple, une seule barre est présente comme le montre la Figure 1.14. Par contre, comme `variableParams` contient un paramètre, `E` qui prend sucessivement les valeurs 1000000, 2000000 et 3000000, trois courbes sont présentes sur le graphe. L'une correspond à la simulation avec le module de Young égale à 1000000, l'autre égale à 2000000 et la troisième égale à 3000000.

1.5.4 Génération de graphes de flexion

Pour les tests de flexion, nous avons réalisé un autre processus de génération. Pour chaque barre et pour chaque simulation, nous superposons les courbes de flexion pour les comparer qualitativement comme le montre la Figure 1.15. Pour cet exemple, nous avons pris un module Young fixe de $1Mpa$. Plus la résolution augmente plus

←T→					id	key	description
☐	Modifier	Éditer en place	Copier	Effacer	1	E	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	2	F	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	3	addDForce	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	4	addForce	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	5	nuY	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	6	nuZ	EMPTY
☐	Modifier	Éditer en place	Copier	Effacer	7	timeStep	EMPTY

FIGURE 1.13 – Table *SofaResult*.

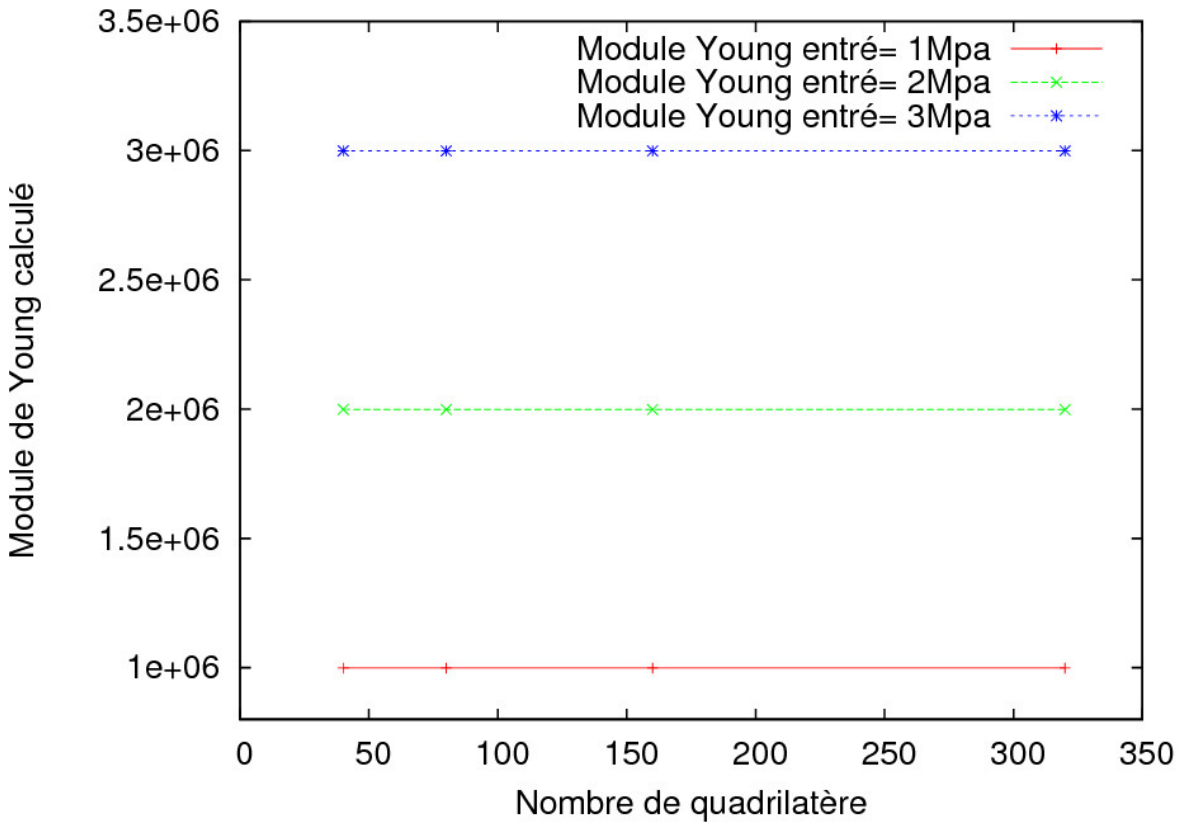


FIGURE 1.14 – Exemple de graphe avec résultat calculé.

la flèche est proche de la solution analytique. L'équation (1.3) correspond à la solution analytique. Cette équation est présente dans la thèse de Vincent Baudet ([4]). Il dit qu'elle est valide pour un coefficient de Poisson nul. L'incompréssibilité n'est pas prise en compte. Cet aspect a été validé auparavant par les travaux de Nesme ([6]). ρ est la masse volumique. Les dimensions de la barre sont explicitées sur la Figure 1.1.

$$y(x) = -\frac{\rho g}{2ETH^3}(6L^2x^2 - 4Lx^2 + x^4) \tag{1.3}$$

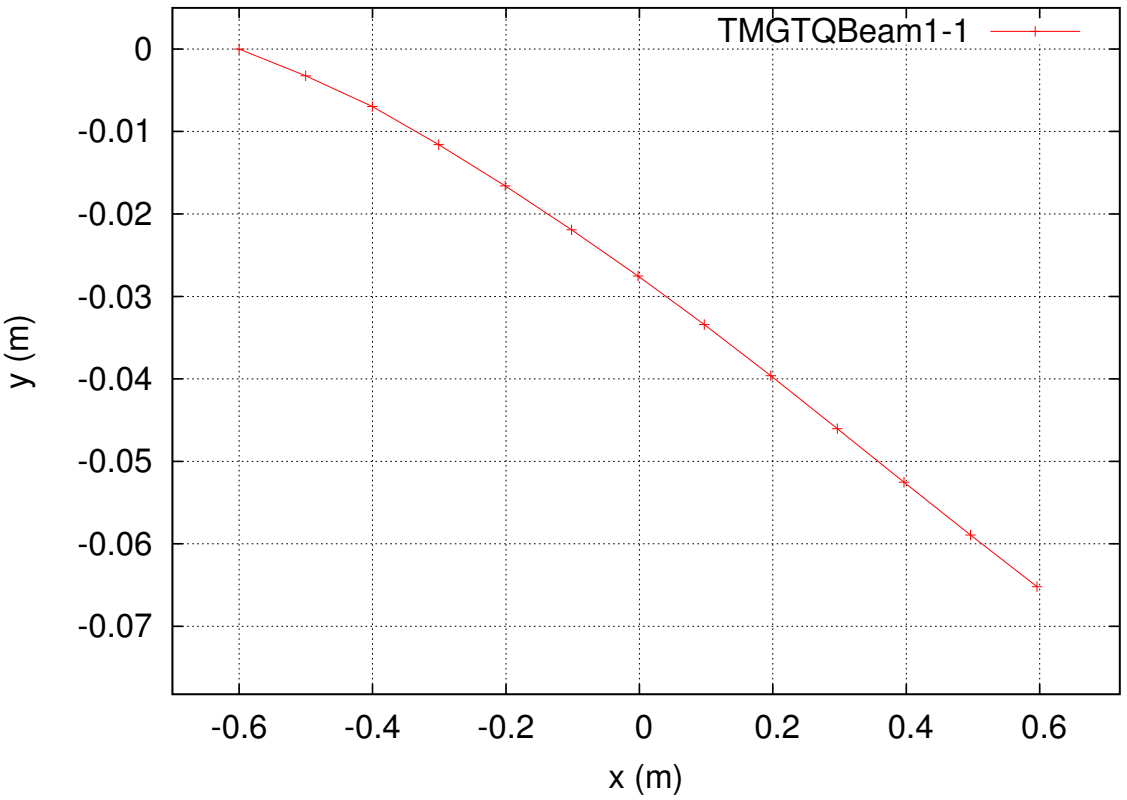


FIGURE 1.15 – Exemple de graphe avec flexion.

Pour avoir un résultat quantitatif, nous avons généré un dernier type de graphe comme le montre la Figure 1.16. Plus un barre est résolue plus elle convergence vers la solution théorique. Ce type de graphe montre l'évolution de l'écart entre toutes les résolutions et la résolution la plus fine. Le dernier point correspond à l'écart entre la plus fine et elle-même, ce qui donne 0.
La Figure 1.17 montre que l'écart est toujours entre une barre 1 de résolution A et une barre 2 de résolution B. La résolution A est plus grossière que la résolution B. L'écart est la moyenne des écarts entre les nœuds de la barre 1 et les nœuds de la barre 2. Pour un graphe d'erreur la barre 2 est la barre de référence qui ne change pas pour toutes les barres 1 du groupe de simulations.
Cet outils accompagné d'une base de données met de l'ordre dans toutes les simulations réalisées, réalise des batteries de test et permet de se concentrer sur l'objectif final : valider l'approche *Méthode des Masse-Tenseurs* comment étant une méthode rapide et précise pour des lois de comportement linéaire et non-linéaire avec intégration implicite quelque soit le type d'éléments utilisé.

1.6 Conclusion

Pour résumer, nous avons montré dans ce chapitre que la librairie *SOFA* permet de faire de la simulation temps réel avec la *MEF* en paramétrant un fichier `xml` décrivant une scène plus ou moins complexe. *SOFA* est une librairie modulaire. Elle sépare tous les composants clés de la simulation.
Nous nous sommes interfacés avec *SOFA* en ajoutant nos propres composants pour les fonctionnalités manquantes comme le calcul des forces internes et leurs dérivées en *Méthode des Masse-Tenseurs* ou l'importation de maillage

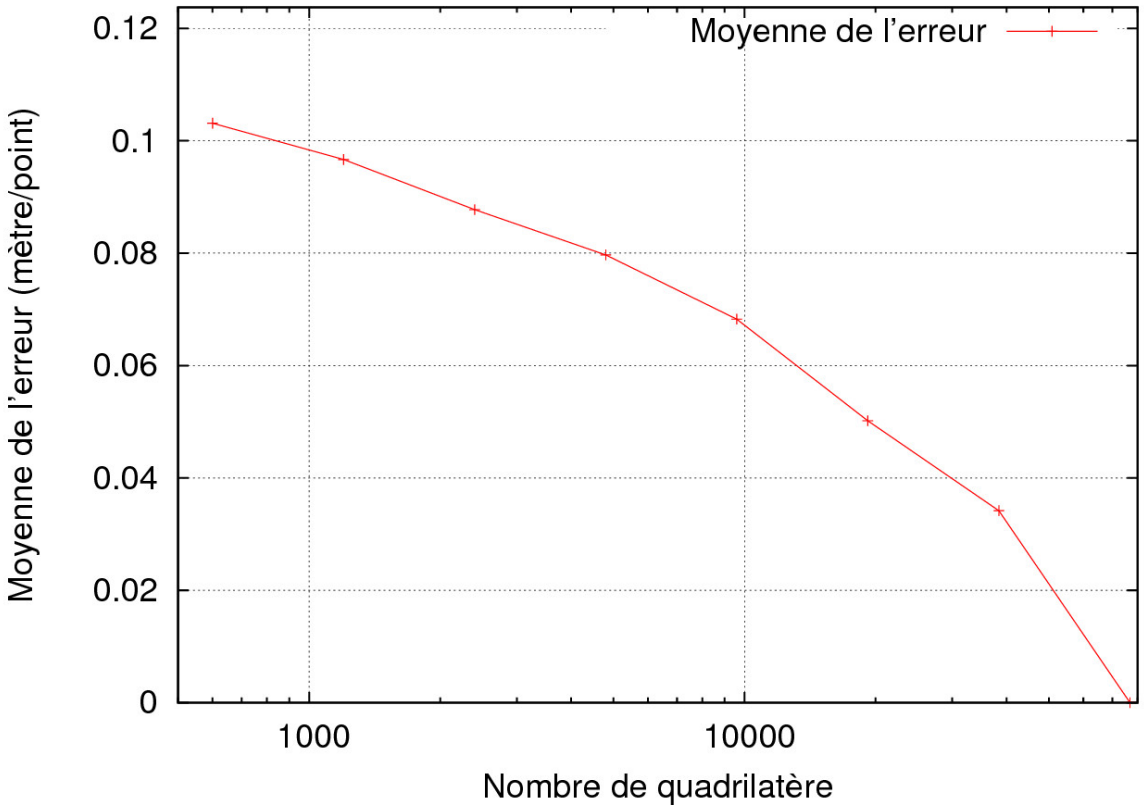


FIGURE 1.16 – Exemple de graphe avec erreur en fonction de la résolution.

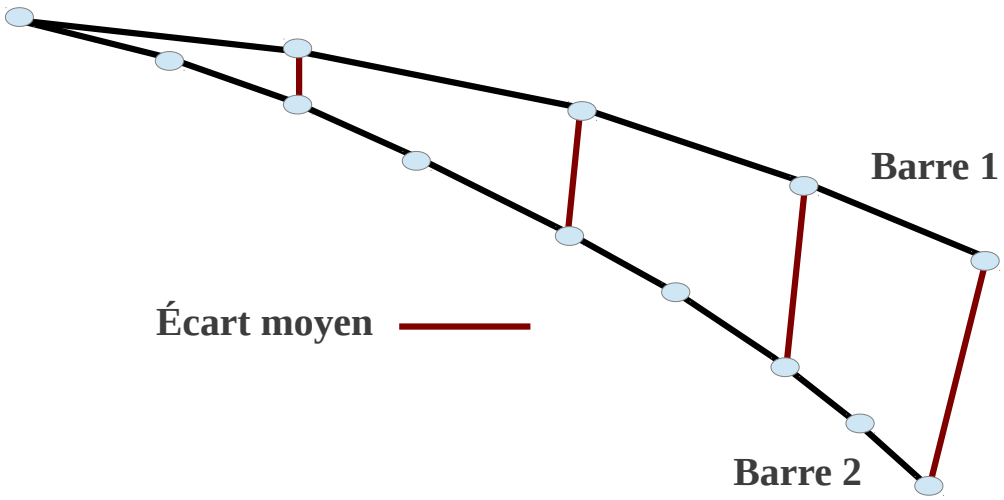


FIGURE 1.17 – Exemple du calcul de l'écart entre 2 barres.

hybride incluant des lois de comportement différentes pour chaque élément.
Nous avons construit une plateforme pour générer des batteries de tests avec plusieurs scénarios incluant la traction et la fléxion. Nous avons organisé les paramètres et les résultats dans une base de données pour classifier les entrées et les sorties et permettre une analyse simple et semi-automatique.
Tous les outils sont en place pour comparer la *Méthode des Masse-Tenseurs* avec la *MEF* en temps et en précision. Avec la *Méthode des Masse-Tenseurs*, nous sommes allés un peu plus loin puisque nous pouvons gérer des maillages hybrides en loi de comportement et type d'éléments. C'est une chose qui est très difficile à faire avec la *MEF* en temps réel.

Références

[1] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni. Sofa an open source framework for medical simulation. In *MMVR'15*, Long Beach, USA, February 2007.

[2] Jérémie Allard, Hadrien Courtecuisse, and François Faure. Implicit fem solver on gpu for interactive deformation simulation. In *GPU Computing Gems Jade Edition*, chapter 21. NVIDIA/Elsevier, September 2011.

[3] Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G Kry. Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics (TOG)*, 29(4) :82, 2010.

[4] Vincent Baudet. *Modélisation et simulation paramétrable d'objets déformables. Application aux traitements des cancers pulmonaires.* PhD thesis, Université Claude Bernard-Lyon I, 2006.

[5] Hadrien Courtecuisse, Jérémie Allard, Pierre Kerfriden, Stéphane Bordas, Stéphane Cotin, and Christian Duriez. Real-time simulation of contact and cutting of heterogeneous soft-tissues. *Medical image analysis*, 2013.

[6] Matthieu Nesme. Modèle déformable par éléments finis-elasticité linéaire et grands déplacements. 2004.

[7] Erik Pernod. Une présentation de sofa en français. *www-sop.inria.fr/asclepios/Publications/Erik.Pernod/Rapport_SOFA_1A.pdf*, 2009.