Du Polymorphisme dynamique au polymorphisme statique : abstraction sans perte de performances

Wilfried KIRSCHENMANN
Senior Consultant

wkirschenmann@aneo.fr

122, avenue du Général Leclerc
F-92100 Boulogne-Billancourt
Tél. : +33 1 46 44 24 66
Fax : +33 1 46 44 28 63

Rencontre C++ Francophone

-

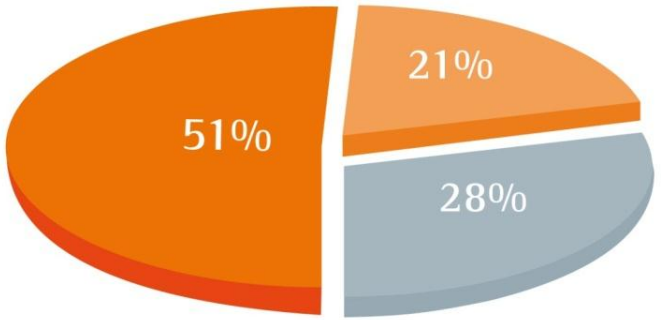
Coverity

Metascale

Sommaire

Fiche d'identité

- ANEO est un Cabinet de Conseil en Organisation et Technologies. Fondé en 2002, la majorité du capital est détenu par des Associés, Exécutifs.
- Son CA 2012 est estimé à 23 M€ pour un effectif de 190 personnes.
- ANEO déploie son expertise dans deux domaines :
  - Business Performance
  - Technologies et Systèmes



Quelques unes de nos références

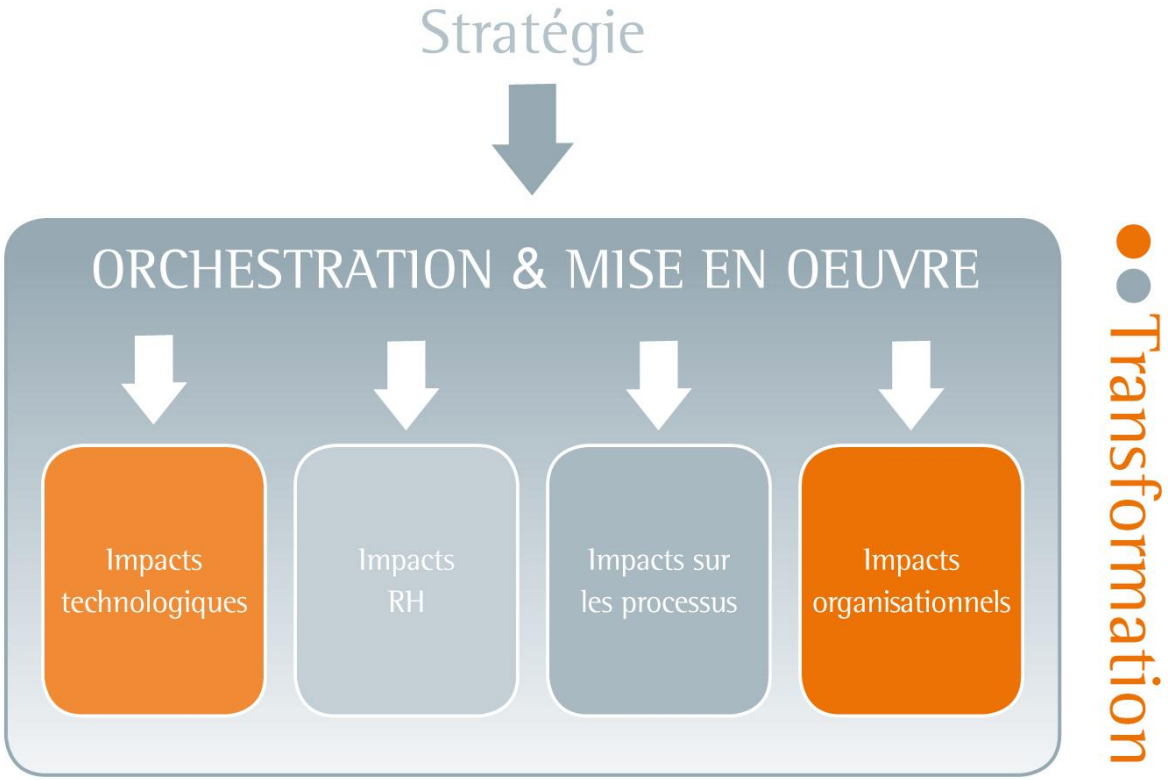
Banque et Finance
Société Générale
Crédit Agricole
Banque de France
NATIXIS
BNP Paribas
HSBC

Assurance
AXA
CNP
Malakoff Mederic
Réunica/Systalians
Mutuelle Nationale et Territoriale

Transports, Energie et Télécom
THALES
SIEMENS
PSA
SNCF
EDF
TOTAL
CEA
ORANGE

De la stratégie à la déclinaison opérationnelle

ANEO accompagne ses clients dans
- la déclinaison opérationnelle de leur stratégie
- la déclinaison des impacts de leur stratégie d'entreprise sur l'organisation et le système d'information.



Les interventions de la practice HPC

RP : HOUCEM HAMZA / 1,9 M€ / SERVICE ET CONSEIL / 19 ETP

- Le domaine du HPC regroupe l'ensemble des expertises impliquées dans la réduction des temps d'exécution des logiciels de calcul

- Nos interventions sont essentiellement orientées sur les activités suivantes :
  - Cadrage de projets HPC
    - Conseil technologique (architectures matérielles, logiciels de calcul, architecture logicielle…)
    - Proposition de scenarii de mise en œuvre de logiciels HPC
  - Implémentation et déploiement
    - Développement
    - Apport d'expertises technologiques
    - Définition et mise en place de procédures de tests et de validation des performances
  - Support et exploitation
    - Centres de services
    - Définition et mise en place de procédures de reporting
  - Amélioration continue
    - Identifier et évaluer les opportunités d'améliorations de performances d'une chaîne de calcul

- Nous intervenons principalement dans le domaine de la BFI où la demande en calcul est forte.

Les expertises de practice HPC

RP : HOUCEM HAMZA / 1,9 M€ / SERVICE ET CONSEIL / 19 ETP

- Mise au point d'un base de benchmarks
  - Implémentation de différents algorithmes
  - Implémentations sur différentes architectures matérielles
  - Implémentations avec différents niveaux d'efforts de développement

  Objectif : pouvoir estimer *a priori* le coût et les gains apportés par un changement d'algorithme, de matériel ou par des optimisations spécifiques

- Etude de la propagation des erreurs d'arrondis dans les code de calcul numérique
  - Analyse de la non reproductibilité des résultats numériques
  - Estimation de la précision des résultats numériques
  - Implémentation d'algorithmes minimisant la propagation des erreurs d'arrondis

  Objectif : garantir la fiabilité d'un résultat numérique

- Outils et méthodes pour la mise au point de codes de calcul multicibles
  - HPCLib : une bibliothèque C++ générique d'algèbre linéaire
  - Mise au point d'une bibliothèque C++ de parallélisation multicible
  - Étude des outils de développement multicibles disponibles sur le marché

  Objectif : assurer la portabilité des performances sur différentes architectures matérielles

Sommaire

Quand l'écriture du logiciel nécessite la mise en œuvre de compétences multidisciplinaires

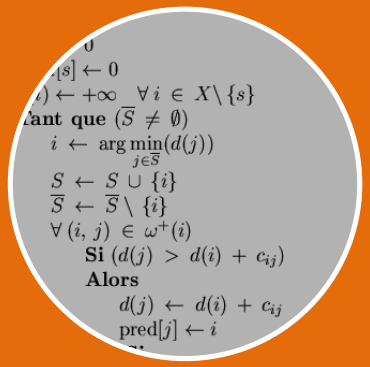
EXEMPLE D'UN LOGICIEL DE SIMULATION PHYSIQUE

Analyse physique

- Description du phénomène
- Mise sous forme d'équations physiques

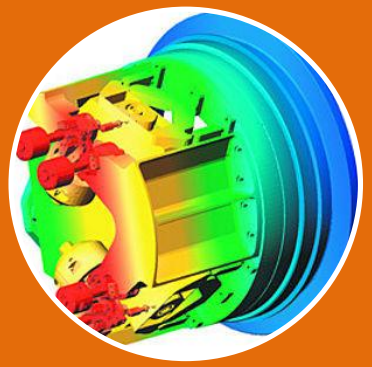
Analyse fonctionnelle ou stochastique

- Caractérisation des propriétés mathématiques des équations mises en œuvre
- Choix des espaces de résolution



Analyse numérique

- Discrétisation et choix des méthodes de résolution



Informatique scientifique

- Caractérisation et visualisation du problème
- Mise en œuvre de l'algorithme de résolution

Les complexités inhérentes à chaque discipline s'ajoutent

LES SOURCES DE COMPLEXITÉ

- Si une modélisation physique n'est pas solvable dans un temps réaliste, les solutions sont :
  - Modifier la modélisation
    - Retirer les termes négligeables
    - Linéariser près du point de fonctionnement
  - Modifier l'espace de résolution
    - résolution de l'équation de la chaleur dans l'espace de Fourrier
    - Utilisation des espaces de Hilbert en mécanique
  - Modifier les méthodes de résolution
    - Résolution itérative plutôt que directe
    - Gauss-Seidel au lieu de Jacobi
  - Modifier l'implémentation
    - Parallélisation
    - Utilisation de bibliothèques

- Chacune de ces solutions à un impact sur les autres éléments de la chaîne

➔ Un tel projet nécessite une organisation et des outils spécifiques

Le compromis entre performances et abstraction

HISTORIQUE DE DIFFÉRENTES SOLUTIONS

Implémentation	$Y \leftarrow \alpha X + Y$	$Y \leftarrow \alpha X + \beta Z + Y$
Langages généralistes (C)	`for(int i=0; i<N; ++i)` `  Y[i]+=a*X[i]`	`for(int i=0; i<N; ++i)` `  Y[i]+=a*X[i]+b*Z[i]`
Bibliothèques de fonction	`axpy(Y, a, X, N)`	`axpypz(Y, a, X, b, Z, N)`
Bibliothèques externes (BLAS)	`blas_axpy(Y, a, X, N)`	`blas_axpy(Y, a, X, N)` `blas_axpy(Y, b, Z, N)`
Langages dédiés (Scilab)	`Y=a*X+Y`	`Y=a*X+b*Z+Y`
ou langages enfouis (Blitz++)	`Y+=a*X;`	`Y+=a*X+b*Z;`

➔ **L'approche des langages dédiés (enfouis) permet de dissocier le traitement des complexités informatiques des complexités mathématiques**

Sommaire

La mise au point d'un langage enfouis (DSEL) est soumise aux possibilités syntaxiques du langage hôte

SÉMANTIQUE DU LANGAGE

Le langage des mathématiques appliquées définit un certain nombre d'opérateurs et de notations et leur accorde une sémantique

f()	Appel de la fonction f
A+B	Addition de A et B
A*B	Multiplication de A par B
A-B	Soustraction dans A de B
A/B	Division de A par B (si défini)
<A,B>	Produit scalaire de A et B
A←B	Affectation à A de B
A_i	Accès au $i^{è}$ élément de A

➔ **Il est impossible de reprendre le langage des mathématiques directement**

La mise au point d'un langage enfouis (DSEL) est soumise aux possibilités syntaxiques du langage hôte

SÉMANTIQUE DU LANGAGE

Un DSEL C++ s'appuie sur la surcharge des opérateur afin de mimer un langage

f()	Appel de la fonction f
A+B	Addition de A et B
A*B	Multiplication de A par B
A-B	Soustraction dans A de B
A/B	Division de A par B (si défini)
dot(A,B)	Produit scalaire de A et B
A=B	Affectation à A de B
A[i]	Accès au i^{è} élément de A

➔ **Un DSEL C++ accorde une sémantique aux opérateur surchargeables en C++**

Une classe définissant un DSEL dédié aux opérations vectorielles de l'algèbre

SÉMANTIQUE DU LANGAGE

```
class Vector {
   public:
      Vector(const int size);
      Vector(const Vector & v)   ;
      ~Vector();
      const Vector& operator=(const Vector & v);
      Vector operator+(const Vector & v) const;
      Vector operator-(const Vector & v) const;
      Vector operator*(const Vector & v) const;
      const float & operator[](int i) const;
      float & operator[](int i);
      int size() const;
   private:
      const int size_;
      float * data_;
};

float dot(const Vector & a, const Vector & b)
```

```
Vector X(N), Y(n), Z(n);
float a, b;
Y = a*X+b*Z+Y;
```

Un code client simple mais quid des problématiques informatiques ? (ex: performances)

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

```
Y = a*X+b*Z+Y;
```

Code généré équivalent

```
Vector tmp1(N);
for(int i=0; i<N; ++i)
   tmp1[i]=b*Z[i];
Vector tmp2(N);
for(int i=0; i<N; ++i)
   tmp2[i]=tmp1[i]+Y[i];
Vector tmp3(N);
for(int i=0; i<N; ++i)
   tmp3[i]=a*X[i];
Vector tmp4(N);
for(int i=0; i<N; ++i)
   tmp4[i]=tmp2[i]+tmp3[i];
for(int i=0; i<N; ++i)
   Y[i]=tmp4[i];
```

5 boucles for et 12 accès mémoire
Au lieu de
1 boucle for et 4 accès mémoire :

```
for(int i=0; i<N; ++i)
   Y[i]=a*X[i]+b*Z[i]+Y[i];
```

Comment encapsuler le corps de la boucle dans un objet pour fusionner les boucles ?

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

```
Y = a*X+b*Z+Y;
```

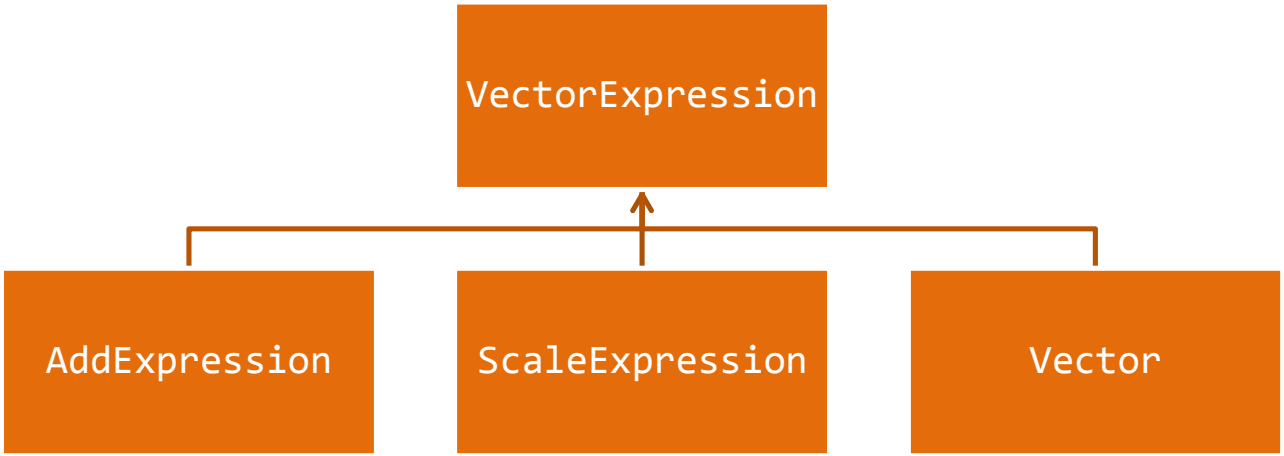

```
Struct VectorExpression {
   const float & operator[](int i) const {
      return a*X[i]+b*Z[i]+Y[i];
   }
   Vector & X, & Y, & Z;
   float a,b;
};

VectorExpression compute_operation = ... ;

for(int i=0; i<N; ++i)
   Y[i] = compute_operation[i];
```

Une arborescence de classes pour représenter les expressions vectorielles

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

Une expression vectorielle peut être :

- La somme de deux expressions vectorielles
- La multiplication d'une expression vectorielle par un scalaire
- Un vecteur



Une arborescence de classes pour représenter les expressions vectorielles

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

Une **expression vectorielle** peut être :

- La somme de deux expressions vectorielles
- La multiplication d'une expression vectorielle par un scalaire
- Un vecteur

```
class VectorExpression {
public:
   virtual int size() const =0;
   virtual const float & operator[](int i) const =0;
};
```

Une arborescence de classes pour représenter les expressions vectorielles

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

Une expression vectorielle peut être :

- **La somme de deux expressions vectorielles**
- La multiplication d'une expression vectorielle par un scalaire
- Un vecteur

```
class AddVectorExpr : public VectorExpression {
public:
   AddVectorExpr (const VectorExpression & l, const VectorExpression & r)
      : l_(l), r_(r) { assert(l_.size == r_.size()); }
   virtual int size() const { return l_.size(); }
   virtual const float & operator[](int i) const { return l_[i]+r_[i]; }

private:
   const VectorExpression & l_;
   const VectorExpression & r_;
};

AddVectorExpr operator+(const VectorExpression& r, const VectorExpression& r)
{
   return AddVectorExpr(l, r);
}
```

Une arborescence de classes pour représenter les expressions vectorielles

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

Une expression vectorielle peut être :

- La somme de deux expressions vectorielles
- **La multiplication d'une expression vectorielle par un scalaire**
- Un vecteur

```
class ScalVectorExpr : public VectorExpression {
public:
   ScalVectorExpr(float a, const VectorExpression & x) : a_(a), x_(x) {}
   virtual int size() const { return x_.size(); }
   virtual const float & operator[](int i) const {return a*x_[i]; }

private:
   const float a_;
   const VectorExpression & x_;
};

ScalVectorExpr operator*(const float & a, const VectorExpression & x)
{
   return ScalVectorExpr(a, x);
}
```

Une arborescence de classes pour représenter les expressions vectorielles

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

Une expression vectorielle peut être :

- La somme de deux expressions vectorielles
- La multiplication d'une expression vectorielle par un scalaire
- **Un vecteur**

```
class Vector : public VectorExpression{
public:
   Vector(const int size) : size_(size), data_(allocate(size_)) {}
   Vector(const VectorExpression & ve) : size_(ve.size()), data_(allocate(size_)){
      for(int i=0 ; i<size_ ; ++i) data_[i]=ve[i];
   }
   ~Vector(){ deallocate(data_); data_ = 0; }
   const Vector& operator=(const VectorExpression & ve){
      if(size_ != ve.size()) reallocate(data, ve.size());
      for(int i=0 ; i<size_ ; ++i) data_[i]=ve[i];
      return *this;
   }
   virtual int size() const { return size_; };
   virtual const float & operator[](int i) const { return data_[i]; }
   float & operator[](int i) { return data_[i]; }

private:
   const int size_;
   float * data_;
};
```

La construction de l'expression vectorielle revient à construire un arbre syntaxique lors de l'exécution

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE L'EXÉCUTION

```
Y = a*X+b*Z+Y;
```

Code généré équivalent

```
ScalVectorExpr bz_(b, Z);
const VectorExpression & bz(bz_);
AddVectorExpr bzpy_(bz, Y);
const VectorExpression & bzpy(bzpy_)
ScalVectorExpr ax_(a, X);
const VectorExpression & ax(ax_);
AddVectorExpr axbzpy_(ax, bzpy);
const VectorExpression & axbzpy(axbz
for(int i=0; i<N; ++i)
   Y[i] = axbzpy[i];
```

Problème :
L'appel à `axbzpy[i]` génère l'appel de 7 fonctions virtuelles :

`axbzpy[i]`	`ax[i]`
`X[i]`	`bzpy[i]`
`Bz[i]`	`Z[i]`
`Y[i]`	

Supprimer les fonctions virtuelle ou construire un arbre syntaxique *à la compilation*

VERS UN LANGAGE PERFORMANT

```
Y = a*X+b*Z+Y;
```

Code généré souhaité

```
typedef ScalVectorExpr<Vector> BZ;
BZ bz(b, Z);
typedef AddVectorExpr<BZ, Vector> BZPY;
BZPY bzpy(bz, Y);
typedef ScalVectorExpr<Vector> AX;
AX ax(a, X);
typedef AddVectorExpr<AX, BZPY > AXBZPY;
AXBZPY axbzpy(ax, bzpy);
for(int i=0; i<N; ++i)
   Y[i] = axbzpy[i];
```

Question : comment construire l'arbre syntaxique de manière automatique ?

Créer une arborescence de classes résolue à la compilation : *Curiously Recursive Template Pattern* (CRTP)

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE LA COMPILATION

Pour pouvoir résoudre les appels de fonctions à la compilation, il suffit de connaître le type réel des objets manipulés :

```
template <class DERIVED> class Base {
public :
   inline void foo() { derived().foo(); }
private :
   inline const DERIVED & derived() const {
      return *static_cast<const DERIVED *>(this);
   }
};

class Derived : public Base<Derived> {
public :
   inline void foo() { ... }
};
```

Résoudre les expressions vectorielles à la compilation

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE LA COMPILATION

On applique le CRTP sur les classes précédemment définies

```
template <class DERIVED>
class VectorExpression {
public:
   inline virtual int size() const =0;
   inline virtual const float & operator[](int i) const =0;

private:
   inline const DERIVED & derived() const {
      return *static_cast<const DERIVED *>(this);
   }
};
```

```
template <class DERIVED>
class VectorExpression {
public:
   inline virtual int size() const {return derived().size(); }
   inline virtual const float & operator[](int i) const {return derived()[i]; }

private:
   inline const DERIVED & derived() const {
      return *static_cast<const DERIVED *>(this);
   }
};
```

Résoudre les expressions vectorielles à la compilation

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE LA COMPILATION

On applique le CRTP sur les classes précédemment définies

```
template <class LEFT, class RIGHT>
class AddVectorExpr : public VectorExpression<AddVectorExpr<LEFT, RIGHT> >{
public:
   AddVectorExpr (const VectorExpression<LEFT>  & l,
                  const VectorExpression<RIGHT> & r)
      : l_(l), r_(r) { assert(l_.size == r_.size()); }
   inline  int size() const { return l_.size(); }
   inline  const float & operator[](int i) const { return l_[i]+r_[i]; }

private:
   const VectorExpression<LEFT>  & l_;
   const VectorExpression<RIGHT> & r_;
};
template <class LEFT, class RIGHT>
AddVectorExpr<LEFT, RIGHT> operator+(const VectorExpression<LEFT>  & r,
                                     const VectorExpression<RIGHT> & r)
{
   return AddVectorExpr<LEFT, RIGHT>(l, r);
}
```

Résoudre les expressions vectorielles à la compilation

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE LA COMPILATION

On applique le CRTP sur les classes précédemment définies

```
template <class VE>
class ScalVectorExpr : public VectorExpression<ScalVectorExpr<VE> > {
public:
   ScalVectorExpr(float a, const VectorExpression<VE> & x)
      : a_(a), x_(x) {}
   inline  int size() const { return x_.size(); }
   inline  const float & operator[](int i) const {return a*x_[i]; }

private:
   const float a_;
   const VectorExpression<VE> & x_;
};
template <class VE>
ScalVectorExpr<VE> operator*(const float & a, const VectorExpression<VE> & x)
{
   return ScalVectorExpr<VE>(a, x);
}
```

Résoudre les expressions vectorielles à la compilation

CONSTRUIRE UN ARBRE SYNTAXIQUE LORS DE LA COMPILATION

On applique le CRTP sur les classes précédemment définies

```
class Vector : public VectorExpression{
public:
   Vector(const int size) : size_(size), data_(allocate(size_)) {}
   template <class VE>
   Vector(const VectorExpression<VE>& ve):size_(ve.size()),data_(allocate(size_)){
      for(int i=0 ; i<size_ ; ++i) data_[i]=ve[i];
   }
   ~Vector(){ deallocate(data_); data_ = 0; }
   template <class VE>
   const Vector& operator=(const VectorExpression<VE> & ve){
      if(size_ != ve.size()) reallocate(data, ve.size());
      for(int i=0 ; i<size_ ; ++i) data_[i]=ve[i];
      return *this;
   }
   inline  int size() const { return size_; };
   inline  const float & operator[](int i) const { return data_[i]; }
   inline float & operator[](int i) { return data_[i]; }
private:
   const int size_;
   float * data_;
};
```

La complexité « informatique » du traitement des expressions vectorielle est localisée et n'impacte pas le code client

POUR ALLER UN PEU PLUS LOIN

Parallélisation multithread

```
const Vector& operator=(const VectorExpression<VE>& ve){
   #pragma omp parallel for
   for(int i=0 ; i<size_ ; ++i) data_[i]=ve[i];
   return *this;
}
```

Vectoriser

```
const Vector& operator=(const VectorExpression<VE>& ve){
   for(sse::Index i=0 ; i<size_ ; ++i)
      sse::Setter(data_,i)=sse::Getter(ve,i);
   return *this;
}
```

Design pattern à employer : command

Utiliser un accélérateur de calcul (GPU)

```
const Vector& operator=(const VectorExpression<VE>& ve){
   cuda::VectorExpressionEvaluate<VE>(ve, *this);
   return *this;
}
```

Nécessite de modifier la conception des classes (tout doit être POD)

Sommaire

L'utilisation de *design pattern* spécifiques permet d'orthogonaliser les complexités issues des différentes disciplines

L'utilisation de DS(E)L permet aux utilisateurs de manipuler les concepts de leur domaine d'expertise

- À chaque expertise son DS(E)L : un DS(E)L de parallélisme peut servir à définir un DS(E)L d'algèbre linéaire puis un DS(E)L d'éléments finis
- Le C++ est un langage à qui permet d'obtenir des DSEL efficaces

Cependant cela ne supprime pas la complexité

- La mise au point d'un DSEL nécessite des compétences informatiques importantes
- La mise au point est souvent plus complexe car les outils ne sont pas adaptés

Gardons espoir : des outils nous aident

- boost::proto est un DSEL C++ … Pour mettre au point des DSEL C++
- Les messages des compilateurs sont de plus en plus compréhensibles
- Des travaux visent à intégrer des outils dédiés aux DSEL dans le langage