Julien Girardin
avec la participation durant le projet industriel de :
Hélène Martorell
Julien Vincent
Quentin Isach
2ème année EN

Rapport de Stage de 2ème année

Objectifs : Réalisation d'un code permettant d'optimiser le produit matrice - vecteur

Table des matières

1 Utilisation générale **3**
1.1 exemple de code 3
1.2 Comment définir un ensemble de points ? 4
1.3 Qu'est-ce qu'un `block` ? 4
1.4 Qu'est-ce qu'un objet `octtree` ? 4
1.4.1 Voisinage d'un `octtree` 5
1.5 Qu'est-ce que l'objet `mat_resultante` ? 6
1.6 Comment déclarer et utiliser un itérateur 6
1.6.1 Utilisation 7
1.7 Remarques 7

2 Utilisation avancée : à destination des futurs développeurs **9**
2.1 Comment déclarer une matrice 9
2.2 Comment déclarer et utiliser les objets `pile` et `filo` 9
2.2.1 qu'est-ce que la structure `pile` ? 9
2.2.2 Qu'est-ce qu'un objet `filo` ? 9
2.2.3 Qu'est-ce qu'un `filo::iterator` ? Et comment l'utiliser 10
2.3 Communication entre C/C++ et FORTRAN 10
2.3.1 L'adressage selon C/C++ 10
2.3.2 L'adressage selon FORTRAN 11
2.3.3 Une meilleur façon d'allouer 11
2.3.4 exemple de passage entre FORTRAN et C/C++ 12
2.4 Comment récupérer les coordonnées d'un point à partir de son numéro ? 13

3 Exploitation des résultats **14**

4 Détails techniques particuliers **16**
4.1 les templates 16
4.2 Iterateur 17
4.3 De la compilation avec g++ 17
4.4 Remarques sur ce document 17

Préface

Tout a commencé avec le projet industriel de deuxième année. Étant interessé, par la programmation depuis assez longtemps, puis plus récemment par Linux, (grace à "Net7" le club informatique de l'ENSEEIHT, très porté sur le logiciel libre) ; le projet proposé par *Jean René Poirier* concernant le developpement de programme compatible avec *GCC, le compilateur de Linux* implémentant une méthode visant à accélérer avec un rapport supérieur à 1000 des produits matrice-vecteurs pour des équations venant de l'electromagnétisme. L'équipe formée par *Hélène Martorell*, *Julien Vincent*, *Quentin Isaach* et moi-même avions pour mission de développer cette nouvelle méthode. Après quelques semaines d'essai de compréhension du problème, nous décidâmes de nous séparer en deux groupes : le premier commencerait la programmation, l'autre continuerait à étudier le problème physique(et programmerait un peu aussi). Il était important pour moi de commencer la programmation le plus vite possible. *Hélène Martorell* étudia le code existant déjà réalisé par *Jean René Poirier* et moi recommençait de zéro un projet, ayant compris rapidemment que je comprendrais le code que si j'étais moi-même confronter aux memes problèmes que *Jean René Poirier* dont le programme était relativement peu clair (il avoua lui-même être assez jeune en programmation C/C++ lorsqu'il commença à coder cette nouvelle méthode (et lui contrairement à nous y arriva)). Ainsi nous continuâmes à coder ce nouveau projet jusqu'à se qu'il devienne plus rapide que le programme d'origine. Il manquait encore de fonctionnalité et nous prîmes pas mal de retard : *Jean-René Poirier* adapta donc le cahier des charges à ce que nous avions déjà réaliser et nous recadra dans un peu sur nos objectifs. La fin du projet industriel arriva... et le projet n'étais pas complétement fonctionnel (normal pour un projet recommencé de zéro), mais notre commenditaire *Jean René Poirier* n'en était pas moins content du travail accompli. La fin de l'année arrivait et je continuais à travailler sur le code, ayant quelques (bonnes) idée pour amméliorer la rapidité du programme. M'y étant beaucoup investi, cela me faisait un peu mal au coeur de lacher ce programme ne sachant pas ce qu'il allait devenir

Après ma demande *Jean René Poirier* accepta ma demande de me prendre en stage et de me faire travailler sur ce projet. Il fallait cette fois-ci intégré le programme dans un autre afin d'en étendre la rapidité (celui-ci possédant déjà les fonctionnalités voulues).

Introduction

Avant de rentrer dans le vif du sujet. Nous allons déjà rappeler brievement les tenants et les aboutissants d'un tel programme. En electromagnétisme la résolution des équations d'objet complexes est un challenge permettant d'améliorer l'ingénérie, ou encore de décrypter des phénomène (provoqué ou non) comme par exemple les radars. La résolution des équations par méthodes inégrales est au programme en $3^{\text{ème}}$ année à l'ENSEEIHT option *Micro-onde*. Le principe de base est de générer la matrice contenant toutes les corrélations entre tous les points. Puis de résoutre l'équation $A \times x = B$. Le but du programme écrit fut d'écrire un structure permettant de regrouper les point proches afin de pouvoir exploiter la relative redondance des données pour en compresser les blocs et accélerer le traitement.

Chapitre 1

Utilisation générale

1.1 exemple de code

```
geo2D* geo = new geo2D((mesh) m); //on construit un objet de geometrie a partir d'un mesh deja existant
block<complex<double> >* p_block; //on va effectuer un assemblage par bloc => on declare un bloc simple de nombre complexe
octtree* division = geo->get_octtree(2); //contruit un octtree de profondeur 2 => voir la suite pour definition de l'octtree
octtree::iterator it1;
octtree::iterator it2;
mat_alt= new mat_resultante<block, complex<double> >(geo); //on instancie une matrice en donnant le type de bloc : bloc simple de nombre complexe
mat_alt->set_algo(COMPRESS); /*#define COMPRESS 1; 1=> active la compression QR; 0=> sans compression*/
mat_alt->set_eps( pow (10, -3) ); /*definit le coeficient d'erreur pour la compression */
for (it1 = *(division->begin(0)); it1.end(); ++it1)
{
  for (it2 = *(division->begin(0)); it2.end(); ++it2)
  {
    if (&(*it1) == &(*it2)) //si l'adresse est la meme cela represente le meme octree
    {
      p_block = mat_alt->prepare_block(&(*it1), NULL);
      p_block->type = DIAG;
      /*ici fonction de remplissage d'un bloc diagonal*/
      mat_alt->ajout_block_diago(p_block);
    }
    else if(neighbour(&(*it1), &(*it2))) //si neihgbour(...) est vrai, les octtrees sont voisins
    {
      p_block = mat_alt->prepare_block(&(*it1), &(*it2));
      p_block->type = NEIGHBOUR;
      /*ici fonction de remplissage d'un bloc voisin*/
      mat_alt->ajout_block_diago(p_block);
    }
    else //sinon les 2 octtrees sont eloignees
    {
      p_block = mat_alt->prepare_block(&(*it1), &(*it2));
      p_block->type = FAR;
      /*ici fonction de remplissage d'un bloc normal*/
      mat_alt->ajout_block(p_block);
    }
  }
}
```

Ce code peut paraître complexe au premier abord, mais il est en réalité assez simple. En effet, il fait appel à de nombreuses possibilités offertes par le langage C/C++ avec pour objectif principal de simplifier l'utilisation finale. La suite décrit tous les objets et leurs emplois. Il peut être utile de se reporter régulièrement au code ci-dessus afin de mieux comprendre les applications des différents objets.

1.2 Comment définir un ensemble de points ?

L'objet `geometrie` contient les informations sur les points de la géométrie. Il existe deux dérivés de cet objet qui sont `geo2D` et `geo3D` et qui correspondent à un objet `geometrie` respectivement en 2 ou 3 dimensions (à utiliser suivant les cas à traiter). Ces dérivés possèdent plusieurs constructeurs afin de s'adapter à d'autres codes. Il est également possible de coder son propre constructeur. Le constructeur de base permet d'ouvrir des fichiers dans lesquels il pourra trouver les données de la distribution de points. Il suffit alors de créer un fichier contenant les coordonnées d'un point par ligne, c'est-à-dire qu'il faut faire un fichier contenant deux ou trois colonnes : la première contiendra les abscisses (x), la seconde les ordonnées (y) et pour une géométrie en 3D une dernière pour la cote (z).
exemple :

```
points avec les coordonnées (x,y,z)
12      12      31
12      6       31
10      10      31
```

Ensuite, afin d'ouvrir ce fichier et de charger la géométrie correspondante, il suffit de déclarer une nouvelle géométrie et de passer en paramètre au contructeur le chemin du fichier.
exemples :

```
1 geo3D* geo = new geo3D("geometrie\0");
2 //mais d'autre solutions sont aussi possibles
3 geo2D geo(struct_mesh);
4 geo3D geo(1000); //cree automatiquement un exemple avec 1000 points
```

Cet objet possède plusieurs méthodes permettant de récupérer les informations sur la géométrie chargée :

- pour connaitre la taille `::get_size_x()`, `::get_size_y()` et, pour `geo3D`, `::get_size_z()` ;
- pour connaitre l'origine `::get_orig_x()`, `::get_orig_y()` et `::get_orig_z()` ;
- pour obtenir un point à partir de son numéro `point* ::get_point(int)`.

1.3 Qu'est-ce qu'un `block` ?

Sans s'attarder, il faut savoir à ce niveau qu'il existe plusieurs types de `block`. Ainsi nous avons pour possibilités :

- le choix entre nombres réels et nombres complexes. Les nombres complexes utilisent deux fois plus de mémoire, il convient donc de les utiliser seulement lorsque cela est nécessaire ;
- le choix entre `block` et `super_block`. Le premier est un bloc simple ne contenant qu'une seul matrice, alors que le second contient quatre matrices pour le calcul sur des diélectriques.

Cette modularité est obtenue en utilisant une programmation par « patron »que l'on appelle *template*. Cette déclaration ne change pas d'une déclaration ordinaire excepté le type de la bloc qui est précisé entre crochets, les paramètres du constructeur venant ensuite. Pour plus d'informations sur les *templates* : consulter le paragraphe correspondant. Ces différents blocs permettent une meilleur utilisation des ressources dans les cas où il n'est pas nécessaire d'utiliser les possibilités offertes. Voici quelques exemples :

```
1 block<double>* p_block //block simple de nombres reels
2 block<complex<double> >* p_block //block simple de nombres
      complexes
3 super_block<double>* p_block //block quadruple de nombres reels
4 super_block<complex<double> >* p_block //block quadruple de nombres
      complexes
5 block<float>* p_block //marche aussi mais il reste des choses a
      coder dans les autres objets
```

1.4 Qu'est-ce qu'un objet `octtree` ?

Un `octtree` est objet listant les points dans une zone de l'espace à 2 ou 3 dimensions. Les fils d'un `octtree` sont les `octtrees` sur les sous-zones définies en divisant par 2 chaque dimensions de la zone de l'`octtree` ascendant. (voir figure 1.1 pour la découpe d'un octtree 3D en $2*2*2 = 8$ fils) Un octtree possède un tableau de pointeur permettant de chainer l'`octtree` à ses fils, ainsi qu'à son ascendant afin de former une structure d'arbre – d'où le nom d'OctTree *Octal Tree* : Arbre Octal pour la cas à 3 dimensions –

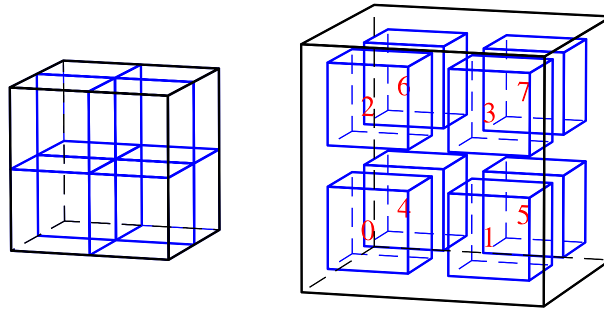

FIGURE 1.1 – représentation schématique de l'`octtree` et de ses 8 fils.

Le niveau d'un `octtree` couvrant la totalité de la geométrie possède le niveau maximal, alors que les `octtrees` de plus petite taille ont par définition le niveau égal à 0. Pour construire un `octtree` de profondeur choisie, on utilise le code suivant :

```
octtree* p_octtree; //pointeur commun pour version 2D et 3D
p_octtree=((geo2D*) geometrie)->get_octtree((int) n1); //version 2D
p_octtree=((geo3D*) geometrie)->get_octtree((int) n2); //version 3D
```

Cela renvoie un `octtree` de niveau $n2-1$. On remarque que l'octtree s'adapte à la geométrie souhaitée : il n'est pas besoin de préciser la dimension de la geométrie utilisée Différentes méthodes sont disponibles sur cet objet :
- la liste chaînée des numéros des points contenus dans l'`octtree` est disponible sous forme de `filo` par la méthode `filo<int>* #::get_listepoint()` ;
- la dimension de l'octtree peut être renvoyée par la méthode `int #::get_dim()`

1.4.1 Voisinage d'un `octtree`

La notion de voisinage d'un `octtree` est très importante pour divers algorithmes de calcul : nous appliquons une méthode différente si les points sont considérés comme proches ou lointains dans l'`octtree` souche. Étant donné que la notion de voisinage de point n'existe pas (et serait trop coûteuse en ressources), il faut une manière d'évaluer la proximité de deux `octtrees`, afin de conserver le caractère géométrique de notre distribution de points. Ainsi, en appelant la fonction `bool neighbour(octtree* oc1, octtree* oc2)`, nous pouvons savoir si deux `octtrees` sont voisins même s'ils ne possèdent pas le même ascendant comme dans l'exemple suivant :

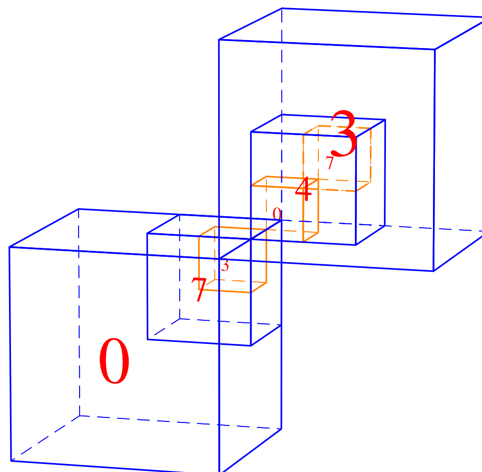

FIGURE 1.2 – représentation schématique de l'`octtree` et de sa numérotation.

L'`octtree` de plus bas niveau (niveau 0) numéroté "0" est voisin à la fois avec l'`octtree` niveau 0 numérotés "7" et "3".

1.5 Qu'est-ce que l'objet `mat_resultante` ?

L'objet `mat_resultante` est l'objet qui va contenir la matrice résultante. Nous rappellons que l'algorithme utilisé ici est un algorithme de calcul par blocs. Ainsi nous ne stockons pas ici la matrice sous forme de tableau de tableaux, mais bien sous forme d'une suite de blocs formant partie ou totalité de la matrice. Cet objet doit interagir avec des blocs, ce qui impose la possibilité d'accepter tous types de blocs. `mat_resultante` possède donc 2 paramètres templates : la type de bloc, et le "scalaire" utilisé pour construire les blocs. Ces deux paramètres composés permettent d'adapter les méthodes de l'objet `mat_resultante` avec le type de blocs à traiter :

- `typebloc* ::prepar_block(octtree* oc1, octtree* oc2)` ;
- `ajout_block(typeblock* bloc)` ;
- `ajout_block_diag(typeblock* bloc)`.

Dans l'exemple suivant, on montre comment déclarer un bloc et une matrice résultante avec de type cohérents cohérente :

```
1 octtree* oc1=une_fonction_pour_avoir_octtree();
2 octtree* oc2=autre_fonction_pour_avoir_octtree();
3 mat_resultante<block, double> mat1;
4 mat_resultante<super_block, complex<double> > mat2;
5 block<double>* pbloc1;
6 super_block<complex<double> >* pbloc2;
7 pbloc1 = mat1.prepare_block(oc1, oc2);
8 mat2.ajout_block(pbloc1); //erreur: type de bloc incoherent
9 mat2.ajout_block(pbloc2); //pas d'erreur. Cela mais attention ajout
      d'un bloc non initialise : possible "segmentation fault"
```

Les deux paramètres templates sont combiné pour obtenir le type de bloc souhaité. Pourquoi ne pas simplement passer un seul paramètre template comme ceci : `mat_resultante< block<complex<double> >` ? Tout simplement parce qu'il nous faut connaitre le "scalaire" utilisé afin de faire un produit matrice vecteur : il aurait fallut rajouter un deuxième paramètre, qui se devait donc d'etre compatible avec le type de bloc. C'est donc dans une optique de simplification que le type de bloc est donné en deux parties.

1.6 Comment déclarer et utiliser un itérateur

Les itérateurs vont permettre de parcourir les `octtrees` des différents niveaux sans avoir à se soucier de parcourir un arbre non complet à un niveau spécifique.

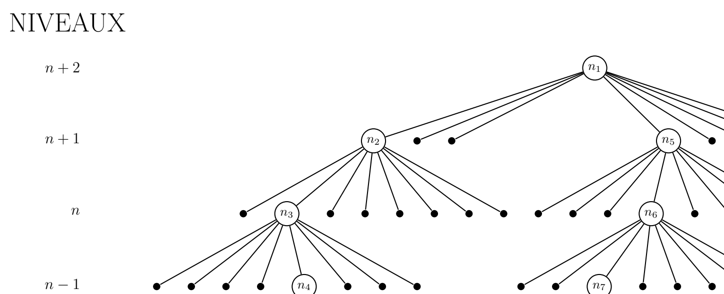

FIGURE 1.3 – arbre d'`octtree`.

Description de l'algorithme : Imaginons que l'algorithme commence en pointant sur n_4. À l'appel de l'`octtree` suivant, voici les différentes étapes du processus :

- il va rechercher dans les fils de n_3 si un `octtree` existe ;
- si oui, il s'arrête ; si non, il remonte d'un niveau et va chercher dans les fils de n_2, il remonte encore s'il ne trouve pas d'`octtree` ;
- il remonte d'autant de niveau que nécessaire ;
- si au dernier niveau (le niveau de la racine), il ne trouve pas d'`octtree`, il passe une variable à `false` indiquant la fin de parcours de l'`octtree` ;
- s'il en trouve un, cela veut dire que cet `octtree` possède au moins un fils ; et en faisant appel à la récursivité : au moins un fils de niveau 0 ou du niveau souhaité ;

- ainsi il va descendre d'un niveau pour rechercher dans les fils des `octtrees` non vide. Il s'arrete dès qu'il trouve le premier `octtree` de niveau recherché.

1.6.1 Utilisation

Pour créer un itérateur, il suffit de le déclarer et de le remplir à l'aide de la méthode :
`octtree::iterator* octtree::begin(int level)`"

```
octtree* oc=fontion_pour_creer_octtree();
octtree::iterator it1;
it1 = *(oc->begin(0));
```

La syntaxe peut paraître surprenante : l'opérateur "=" utilisé sur un objet est souvent signe de mauvaise surprise. Ici, l'utilisation du "=" est faite en surchargeant l'opérateur et en effectuant une copie des informations contenues dans l'objet itérateur. Différents opérateurs et méthodes sont définis sur les itérateurs :

- l'opérateur `++` à gauche : permet de faire pointer (en interne) l'itérateur vers le prochain `octtree` non nul de niveau choisi lors de la construction ou la copie de l'itérateur ;
- l'opérateur `*` à gauche : permet d'obtenir le pointeur de l'`octtree` indiqué par l'itérateur ;
- la méthode `bool ::end()` renvoie `false` s'il ne reste plus d'`octtree` à parcourir.

Ceci amène à une syntaxe simple de boucle *for* :

```
octtree::iterator it1;
octtree::iterator it2;
for (it1 = *(octtree->begin(0)); it1.end(); ++it1)
{
  for (it2 = *(octtree->begin(0)); it2.end(); ++it2)
  {
  ///code ici////
  }
}
```

Nous remarquons à nouveau l'opérateur "=" sur les itérateurs $it2 = *(octtree- > begin(0))$ qui permet d'initialiser l'`octtree` à partir d'un `octtree` pointant sur le début de l'arbre. Cette syntaxe avec deux itérateurs permet de parcourir l'arbre et d'avoir toutes les possibilités de couple d'`octtrees`. Dans le cas d'une matrice symétrique, une autre structure permet de définir (et de calculer) que la moitié de la matrice.

```
octtree::iterator it1;
octtree::iterator it2;
for (it1 = *(octtree->begin(0)); it1.end(); ++it1)
{
  for (it2 = it1; it2.end(); ++it2)
  {
  ///code ici////
  }
}
```

Nous remarquons encore l'opérateur "=" sur les itérateurs $it2 = it1$ qui permet de copier la structure d'un itérateur dans un autre. Cette syntaxe avec deux itérateurs permet de parcourir l'arbre et d'avoir toutes les possibilités de couple d'`octtree` **sans doublon**.

1.7 Remarques

Le calculs des différents blocs est assuré par des fonctions externes à l'objet `mat_resultante` afin de permettre une meilleur adaptation à de multiples code de calculs physique (et donc éviter d'avoir à redéfinir des méthodes à l'interieur de l'objet `met_resultante` lui-même).

Pour éviter de problèmes de mémoire et facilité le travail de programmtion des codes physique c'est l'objet de type `mat_resultante` lui même qui fait m'allocation mémoire et le placement des listes de points respectivement aux bonnes dimensions et bonnes valeurs.

Dans la boucle de caluls on détermine quel type de bloc il vient à calculer. Ceci pourrait de même être fait à l'interieur de l'objet lui-même, mais cela empecherait l'utilisateur de changer de codes suivant

les cas (où même de définir d'autre cas si l'humeur lui en prend). Ainsi la détermination des choix est faite au niveau de l'utilisation finale pour afin de faciliter les modifications.

Chapitre 2

Utilisation avancée : à destination des futurs développeurs

2.1 Comment déclarer une matrice

En regardant, la déclaration des objets et des méthodes du projet nous pouvons remarquer que les matrices sont très utilisées. L'objet `matrice` est une classe réalisée à l'aide *template*. Ainsi la déclaration et l'utilisation de matrice contenant des entiers ou des nombres à virgule flottante se font sans ajout de code supplémentaire.

Dans l'exemple suivant, nous déclarons une matrice de type `int` :

```
matrice<int>* mat;
mat = new matrice<int>(dim_1, dim_2);
```

La déclaration est une déclaration avec *template* comme déjà évoqué. L'utilisation se fait dès lors comme une matrice traditionnelle en C/C++ (tableau de tableaux) grâce à la définition de l'opérateur « `[]` ». Celui-ci permet d'écrire le plus naturellement possible : `p_matrice[j2][j1]` afin d'accéder à un élément de cette matrice.

2.2 Comment déclarer et utiliser les objets `pile` et `filo`

2.2.1 qu'est-ce que la structure `pile` ?

Nous ne parlerons que très brievement des piles car il n'est pas nécessaire de connaître cette structure afin d'utiliser les objets `filo`. Pour une utilisation plus avancée (comprendre programmation de nouvelles méthodes pour cet objet), il est fortement recommandé d'étudier attentivement le lien entre l'objet `filo` et la structure `pile`.

Une pile est une liste chaînée d'éléments se terminant par une référence de valeur `NULL`. Dans cette structure nous trouvons deux attributs :

- l'attribut `valeur` qui correspond à l'élément stocké dans ce maillon de la liste chaînée ;
- l'attribut `suivant` qui correspond à un pointeur vers le maillon suivant de la liste. Ce pointeur adopte automatiquement le type de pointeur souhaité.

Grâce aux *templates* nous pouvons définir tous types de `pile` :

```
pile< matrice<int>* > maillon_pile;
```

Dans l'exemple précédent le type de `pile` defini est lui-même comme un type utilisant les *templates* : cela montre que nous pouvons facilement imbriquer des *templates*.

2.2.2 Qu'est-ce qu'un objet `filo` ?

Un objet `filo` est un objet associé à la structure *pile*, ainsi, il est un objet de contrôle des listes chaînées `pile` qui permet d'ajouter un élément en tête de liste (méthode `push(type element)`) ou d'enlever un élément en le retournant (méthode `type pop()`) : ceci caractérise le comportement d'une pile *FILO* (*First In Last Out*). Cet objet contient un pointeur vers le premier élément de la liste chaînée `pile` qui lui est associée et que l'objet peut contrôler :

– en collectant un pointeur vers la pile avec la méthode `::get_pointeur()` ;
– en collectant la taille de la pile avec la méthode `::size()` ;
– en ajoutant un élément en tête de liste avec la méthode `::push()` ;
– en supprimant un élément en tête de liste avec la méthode `type ::pop()`.

2.2.3 Qu'est-ce qu'un `filo::iterator` ? Et comment l'utiliser

Nous lui avons ainsi associé l'iterateur : l'objet `filo::iterator` permet de contrôler l'objet `filo`. Voici un exemple de code :

```
1 double somme=0;
2 filo<double> p_liste;
3 /*ici, remplissage et utilisation de la liste*/
4 filo<double>::iterator it(p_liste); //declaration et initialisation de l'iterateur
5 while(it.end()) //tant qu'on est pas arrive a la fin de la liste : it.end() vaut vrai
6 {
7   somme = somme + (*it); //effectue la somme des nombres de la liste : (*it) est l'element de la
        liste, ici un double
8   ++it; //permet de faire pointer l'iterateur sur le nombre suivant
9 }
```

Remarque : Les itérateurs sont aussi bien adpatés à la structure de boucles *for* que de boucles *while* comme le montre l'exemple précédent.

2.3 Communication entre C/C++ et FORTRAN

2.3.1 L'adressage selon C/C++

En langage C/C++, l'allocation d'une matrice peut-être faite de la façon suivante :

```
1 int** p_matrice;
2 p_matrice = new int*[dim_1]
3 for (int i=0; i<dim_1; i++)
4 {
5   p_matrice[i] = new int[dim_2];
6 }
```

Dans ce morceau de code, le pointeur `p_matrice` pointe sur une zone mémoire de taille : $dim_1 \times sizeof(pointeur\ int*)$. Le reste de la mémoire est alloué en zones de taille : $dim_2 \times sizeof(int)$, mais **ne sont pas forcément contiguës**. Quand on accède à un élément de la matrice :

`p_matrice[j1][j2]`

L'adressage se fait suivant le schéma ci-dessous :

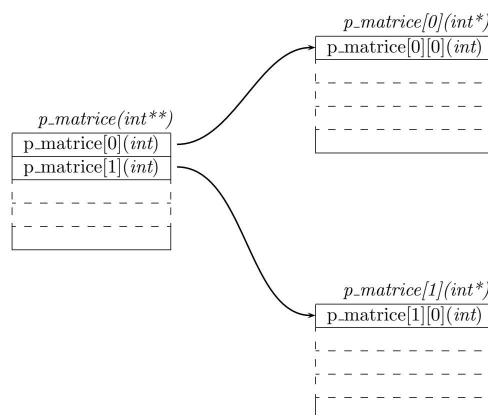

FIGURE 2.1 – schéma d'adressage classique en programmation C/C++.

Ainsi, comme on peut le voir sur le schéma précédent, il n'existe pas vraiment d'objet matrice formant une unité. C'est l'assemblage de plusieurs morceaux qui donnent l'*impression* d'avoir une matrice. De plus chaque allocation et désallocation prennent beaucoup de temps à l'échelle de ce type de programme ; il est donc mal à propos d'utiliser un tel type d'allocation.

2.3.2 L'adressage selon FORTRAN

L'adressage en FORTRAN est faite de façon différente. Contrairement au langage C/C++, le concept de matrice existe dans ce langage. Le compilateur alloue la mémoire pour l'ensemble des éléments, c'est ensuite qu'il associe les coordonnées (couple de deux entiers) à un emplacement dans la mémoire suivant le schéma ci-dessous :

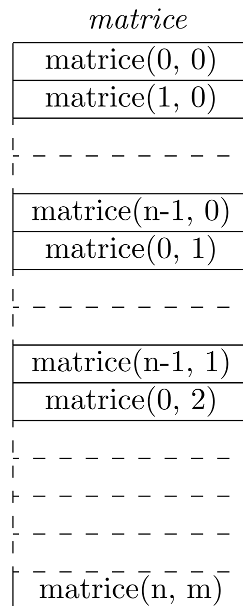

FIGURE 2.2 – occupation mémoire d'une matrice en FORTRAN

2.3.3 Une meilleur façon d'allouer

Il existe cependant une possibilité en C/C++ permettant d'avoir un adressage compatible avec celui du FORTRAN, et ainsi de pouvoir passer un objet d'un langage à l'autre.L'astuce consiste à allouer manuellement un vecteur faisant la taille totale de la matrice.Pour garder la même utilisation des matrices en C, on peut allouer comme à l'habitude un tableau de pointeur. Nous pouvons remplir ce tableau de pointeur afin d'avoir retro-compatibilité totale avec du code C/C++ et l'utilisation classique de `p_matrice [j1][j2]`, à l'aide du code suivant :

```
1  T* data = new T[dim_1*dim_2];
2  /*on alloue la totalite de la matrice*/
3  p_matrice = new T*[dim_1];
4  /*on aloue un vecteur de vecteurs*/
5  for (i=0; i< dim_1; i++)
6  {
7    p_matrice[i]=(data + i*dim_2);
8    /*on remplit les pointeurs*/
9  }
```

FIGURE 2.3 – code d'allocation amélioré d'une matrice en C/C++

Ce code fonctionne quelque soit le type `T` à allouer. Il faut noter qu'il n'est pas nécessaire de garder une variable contenant l'adresse du début de la zone mémoire de la matrice(dans l'exemple `data`), car le premier pointeur : $p_matrice[0]$ contient cette même adresse et est déjà du type `T*`

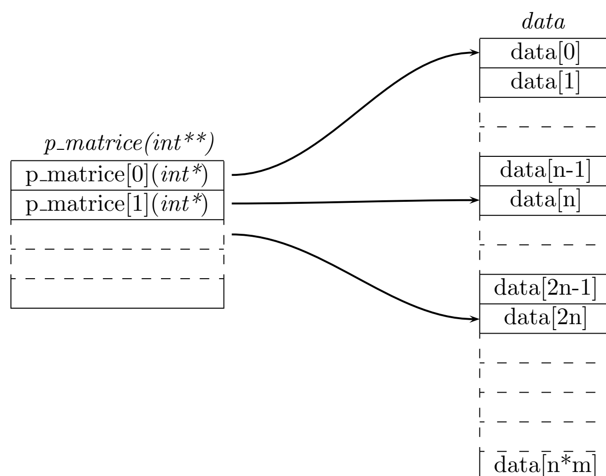

Figure 2.4 – version améliorée de l'allocation d'une en C/C++

Le passage de C/C++ à FORTRAN nécessite la version au moins la version 90 de FORTRAN, afin de bénéficier des pointeurs dans le langage. L'appel de la routine

```
call C_F_POINTER(P_MATRICE,Z,(/dim_1,dim_2/))
```

permet de transformer le pointeur C/C++ précédemment décrit (en le castant si necessaire, voir remarques suivantes), en une matrice FORTRAN que l'on récupère dans `Z`.

Remarques :

- attention, il reste cependant une différence importante : **la matrice est transposée**. En effet on remarque qu'en FORTRAN les éléments `matrice(i,j)` et `matrice(i+1,j)` se succèdent. En C le premier déférencement de pointeur, permet de le déplacer d'une dimension et ce sont donc les elements `p_matrice[j][i]` et `p_matrice[j][i+1]` qui se succèdent.
- on peut dépasser la taille de la dimension correspondante lors du déplacement avec le deuxième indice : on va sur la ligne suivante si ce n'est pas la dernière. Il n'y a pas d'erreur de segmentation levée par la système. Ceci peut-être vu comme un bien, mais cela peut aussi amener à des erreurs de programmations non détectées.
- Pour passer des pointeurs évolué il est peut être parfois necessaire de caster sur un type simple avant de passer le pointeur à du code FORTRAN. Par exemple pour le type `complex<double>` en C/C++, on vérifie d'abord qu'on accéde à la partie réelle sur les 8 premiers octets après un pointeur `complex <double>`, puis à la partie imaginaire à la suite de ses 8 octets. Ainsi il est possible d'effectuer un castage en type `double` comme ceci : `double* Z = (double*)(*(p_block->mat))[0];`
- Il faut cependant noter que l'allocation dynamique n'est pas la même entre FORTRAN et C/C++, il faut donc eviter de désallouer dans un langage ce qui a été alloué dans un autre language.

2.3.4 exemple de passage entre FORTRAN et C/C++

Exemple de code à mettre à la place de `/*ici fonction de remplissage d'un bloc ...*/` dans le code général en début de rapport.

```
1  mat_temp.n =p_block->nb_points1;
2  mat_temp.m =p_block->nb_points2;
3  l.in = p_block->liste2;
4  l.out = p_block->liste1;
5  l.n_in = p_block->nb_points2;
6  l.n_out = p_block->nb_points1;
7  mat_temp.Z = (double*)(*(p_block->mat))[0];
8  mat_temp2.Z = (double*)(*(p_block->mat2))[0];
9  mat_temp3.Z = (double*)(*(p_block->mat3))[0];
10 mat_temp4.Z = (double*)(*(p_block->mat4))[0];
11 assemblediel_bloc(&mat_temp, &mat_temp2, &mat_temp3, &mat_temp4, &par,&m,d,thetarad,mu, &l);//
       appel de la fonction fortran
```

Dans cet example, on voit tous les paramètres passés au fortran :

– l est une structure qui contient le nombre et la liste des éléments en entrée et en sortie.
– mat_temp, mat_temp2, mat_temp3 et mat_temp4 sont des tableaux contenant respectivement les matrices (de type complex<double>) p_block->mat, p_block->mat, p_block->mat2, p_block->mat3 et p_block->mat4. Avant l'appel de la fonction assemblediel_bloc les matrices sont vides et non inialisées, mais l'espace est réservé au sein du système d'exploitation. Le passage d'une matrice à un vecteur se fait en "castant" le premier élément.(Rappel : une matrice<complex<double> > est assimilable à un poiteur complex<double>**, donc le premier élément (qui donne le premier élément de la matrice) est de type complex<double>*. Ce tableau est ensuite converti par "casting" en un tableau de type double* possèdant des éléments deux fois plus petits (2*sizeof(double)=sizeof(complex<double>)) mais en nombre deux fois plus important.
– seul l'attribut Z de mat_temp (valable pour les autres) est utilisé. On aurait pu faire l'économie d'un structure mais par souci de simplicité (changer tout le code pour faire disparaitre cette structure). Elle est resté.

2.4 Comment récupérer les coordonnées d'un point à partir de son numéro ?

Ce qui va être stockée, pour plus de facilité ce sont des numéros relativement aléatoires qui ont été attribués à chaque point. Pour ne pas perdre l'information sur les coordonnées, la correspondance entre le numéro du point et ses coordonnées a été stockées dans la géométrie. Pour récupérer les coordonnées dans une structure de type point, il suffit de déclarer un point et d'appeler la methode get_point(int) de la géométrie.

exemple :

```
1 point *pt1;
2 pt1 = geo->get_point(i);
```

Attention, la fonction get_point renvoit un pointeur sur le point.

Chapitre 3

Exploitation des résultats

Quelques résultats peuvent être établis avec des tests sur le programme (voir section sur la mise en place des tests).

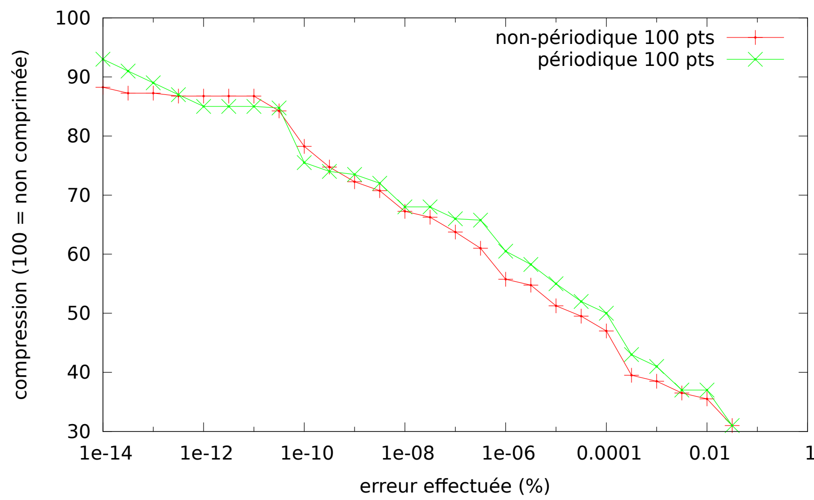

FIGURE 3.1 – différence pour 100 points

Le test sur 100 points montre une forte similitude au niveau de la compression entre cas périodique et cas non-périodique. Il est cependant insuffisant d'utiliser 100 points pour faire de la compression, comme on le voit les courbes ne descendent pas en dessous de 30%

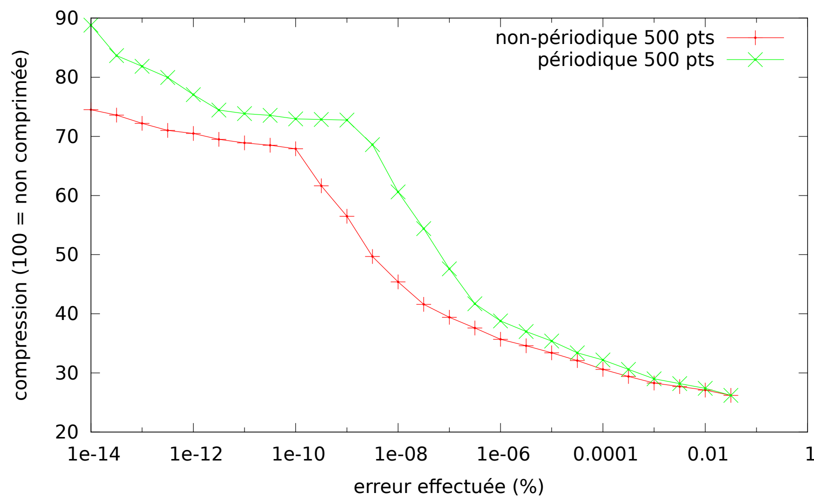

FIGURE 3.2 – différence pour 500 points

Comme nous pouvons le voir sur le graphique ci-dessus, un écart se creuse avec 500 points. Il apparait une pente plus importante autour de la précision 10^{-7}.

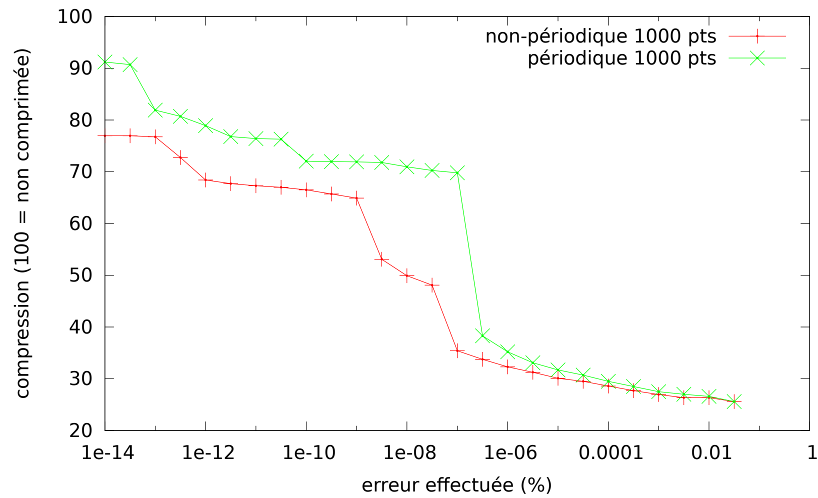

Figure 3.3 – différence pour 1000 points

Cette pente plus importante est encore accentué avec 100 points. Cette pente correspond à la précision numérique de l'algorithme. Bien que basé sur des templates qui autorisent un changement de type de façon très simple, la double précision possède tout de même la limite des flottant classiques.

Chapitre 4

Détails techniques particuliers

4.1 les templates

Extrait de Wikipedia sur la "Métaprogrammation avec des patrons" :
Avantages et inconvénients :

- **Compromis entre la compilation et l'exécution** : comme tout le code sous forme de patron est traité, évalué et généré à la compilation, la compilation prendra plus de temps tandis que le code exécutable pourra être plus efficace. Bien que cet ajout soit généralement minime, pour des gros projets, ou pour des projets qui utilisent des patrons en grande quantité, cela peut être important.
- **Programmation générique** : la métaprogrammation à base de patrons permet au programmeur de se concentrer sur l'architecture et de déléguer au compilateur la génération de n'importe quelle implémentation requise par le code client. Ainsi, la métaprogrammation à base de patrons peut être accomplie via du code vraiment générique, facilitant la minimisation et la maintenabilité du code.
- **Lisibilité** : la syntaxe et les idiômes de la métaprogrammation à base de patrons sont ésotériques par rapport à du C++ conventionnel, et des métaprogrammes avancés peuvent être très difficiles à comprendre. Les métaprogrammes peuvent ainsi être difficiles à maintenir pour des programmeurs peu rompus à cette technique.
- **Portabilité** : à cause de différences dans les compilateurs, le code reposant fortement sur de la métaprogrammation à base de patrons (en particulier les nouvelles formes de métaprogrammation) peuvent avoir des soucis de portabilité.

déclaration de classes Quelque exemple sur la déclaration avec des templates :

```
template <class T>
class mypair
{
    T values [2];

  public:
    mypair (T first, T second);
    int getmax();
};
template <class T>
mypair::mypair (T first, T second)
{
  values[0]=first; values[1]=second;
}
```

```
template <class T>
T mypair::getmax()
{
  if (valeur[0]>valeur[1])
  {
    return values[0];
  }
  else
  {
    return values[1];
  }
}
```

instanciation d'un objet à base de template

```
mypair<int> mon_objet(115, 36);
```

ici la <class T> précédemment mise en "variable" va être remplacée et recompilée en prenant "int" comme classe (un type de base est dans ce cas là considéré aussi comme une classe)

4.2 Iterateur

il existe une classe iterateur dans la librairie standard qui permet de fournir un objet standardisé à destination de la programmation d'iterateur. Ceux-ci ne sont pas utilisés car les iterateurs ne permettent ici que de se déplacer vers l'élément suivant, et donc auraient sous utilisé les attributs de la classe. De plus la structure d'arbre aurait obligé d'en rajouter.(à reformuler)

4.3 De la compilation avec g++

g++ est un des rares compilateur implémentant complétement les templates. Le code n'a pas été testé avec d'autre compilateur que g++. L'instanciation des template est automatique, pour cela g++ compile 2 fois chaque fichier :

- une fois en marquant les templates dont il a besoin
- une seconde fois pour compiler les templates dont il a eu beoin la première fois

Note : le code nécessite g++ >= 3.0 pour compiler.

4.4 Remarques sur ce document

Ce document a été entièrement réalisé avec des logiciels libres sous le système d'exploitation « Linux » libre lui aussi. Voici un détail sur les différents logiciels utilisés (tous libres) :

- la distribution Ubuntu *Lucid Lynx* a été utilisée pour tous les programme ci-dessous ;
- ce document a été écrit en LaTeX, et compilé avec `pdflatex`, version 2009 issu des paquets Ubuntu ;
- les illustrations sont réalisées à l'aide de `PSTricks` et de son extension `pst-solides3d` ;
- les graphiques sont réalisés avec *Gnuplot* version 4.4 ;
- le programme qui a servi à faire les tests et à récupérer les résultats est écrit en *Python* ;
- un *GNU Make* a été utilisé pour faciliter la compilation du document.
- un dépot de code de type *Mercurial* a été utilisé.(utiliser *hg clone http ://hg.jugirardin.fr/bem2dqr/* pour avoir une copie du code du programme, ainsi que le code de ce document)