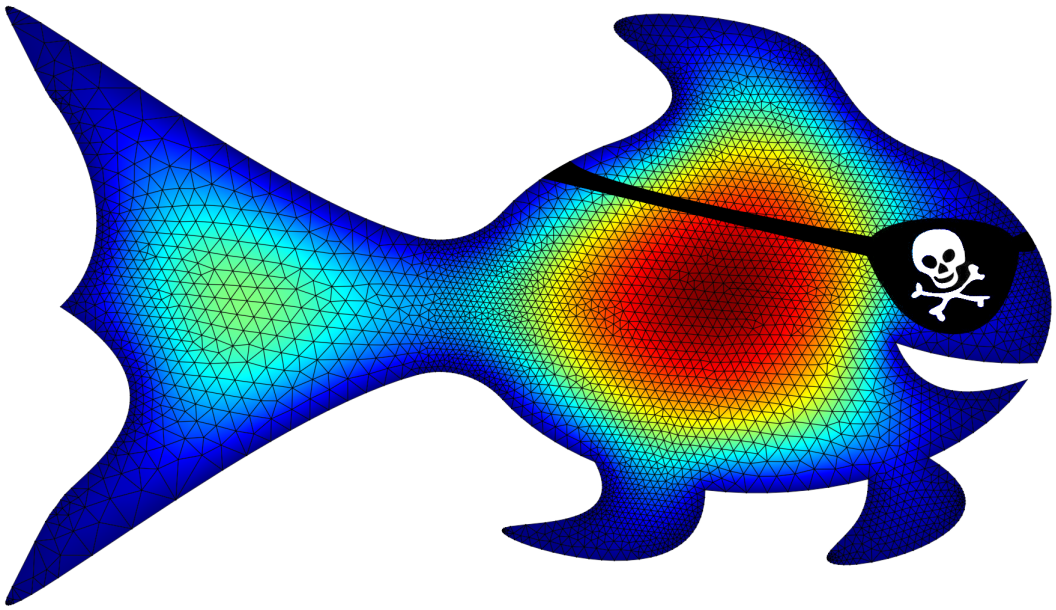
Finite elements for dummies : Band solver

Louis Devillez
Jérome Eertmans

Ce document a pour but d'aider les étudiants du cours d'Éléments Finis, donné par MM. Vincent Legat et Jean-François Remacle à l'Ecole Polytechnique de Louvain, en leur offrant une solution détaillée aux devoirs.



1 Le problème à résoudre

Cette semaine, l'objectif du devoir va être de tirer profit du caractère "bande" de votre système afin de le résoudre bien plus rapidement. Pour ce faire, vous allez devoir éditer 4 fonctions :

1. Une fonction qui renumérotera vos noeuds de manière intelligente (ou non)

```
void femMeshRenumber(femMesh *theMesh, femRenumType renumType);
```

2. Une fonction calculant la largeur de bande de votre système

```
int femMeshComputeBand(femMesh *theMesh);
```

3. Une fonction qui assemble le système en tirant profit du caractère symétrique de ce dernier

```
void femBandSystemAssemble(femBandSystem* myBandSystem, double *Aloc, double *
    Bloc, int *map, int nLoc);
```

4. Une fonction qui résout le système bande avec une élimination gaussienne

```
double *femBandSystemEliminate(femBandSystem *myBand);
```

2 Optimiser la mémoire

Dans un précédent devoir, on vous a expliqué comment allouer un bloc continu en mémoire, ce dernier pouvant s'apparenter à un tableau contenant des éléments de même type.

Comme à peu près tout en programmation, l'accès à un endroit spécifique en mémoire a un coût. Ce dernier est la conséquence du temps que prend le signal à transiter entre votre CPU et votre mémoire, et donc aussi de la rapidité de lecture ou écriture de votre mémoire. Pour réduire le temps d'exécution, il faudrait donc réduire le nombre d'appels à la mémoire...

2.1 Le cache à la rescousse !

Afin de palier au problème qu'est le coût des appels mémoires classiques, des gens très intelligents ont inventé la mémoire cache. Votre ordinateur en possède très certainement et cette dernière est souvent découpée en plusieurs niveaux (L1, L2, L3, etc.) en fonction de leur taille et leur rapidité d'accès. Une mémoire L1 sera souvent très rapide et proche du CPU, mais de petite taille car son coût de fabrication est très élevé. Votre disque dur ou SSD fait certainement plusieurs Go, alors qu'une mémoire cache se chiffre plutôt en quelques Mo.

Le cache sert donc à mémoriser les données récentes, dans l'espoir qu'elles vont être réutilisées prochainement pour gagner du temps en évitant un accès mémoire très long.

Définiton : On définit alors le *hitrate* (taux de réussite) d'un cache comme étant le nombre de fois où on a vraiment réutilisé une donnée située dans le cache, sur le nombre total de fois où on a été demander à notre cache si il possédait cette donnée.

2.1.1 En pratique

En réalité, le cache ne sauvegarde pas uniquement la donnée que vous avez demandée mais plutôt le résultat de votre requête à votre mémoire. Ceci se présente souvent sous la forme d'un bloc continu dans lequel votre donnée se trouve.

Prenons le cas d'un accès à l'élément `5` de votre tableau `A = {4, 7, 12, 8, 9, 10, 1, 99, 33}`. Au lieu de vous renvoyer juste `A[5]`, la mémoire va vous envoyer un plus grand bloc, par exemple `{A[4], A[5], A[6], A[7]} = {9, 10, 1, 99}`, duquel votre programme va extraire automatiquement la valeur `10`.

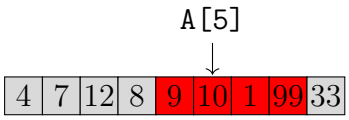


FIGURE 1 – Example du *caching* d'une partie de `A` quand `A[5]` est requis.

2.1.2 Exemple simple mais efficace

Pour voir l'effet du cache sur la rapidité de votre code, un exemple très connu est celui de la multiplicaton matricielle : $\boldsymbol{C}^{m\times l} = \boldsymbol{A}^{m\times n} \times \boldsymbol{B}^{n\times l}$. En notation indicielle, on obtient :

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j} \tag{1}$$

Pour créer une matrice, l'un serait tenté de faire comme suit : allouer une tableau de tableaux, où chaque sous-tableau est une ligne de la matrice. Voici un exemple de création d'une matrice de zéros :

```
double** zerosMatrix(int m, int n) {
  double **M = malloc(sizeof(double*) * m);
  for (int i = 0; i < m; i++) {
    M[i] = calloc(sizeof(double), n); // Allocation d'une ligne
  }
  return M;
}
```

Avec cette représentation, il est possible d'accéder à $A_{i,j}$ en faisant `A[i][j]` et on pourrait donc implémenter une multiplicaton matricielle de la manière suivante :

```
double** ijk(double **A, double **B, int m, int n, int l) {
  double **C = zerosMatrix(m, l);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < l; j++) {
      for (int k = 0; k < n; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
   }
  return C;
}
```

Ici, les lignes ne sont pas continues entre-elles dans la mémoire, ce qui signifie que la mémoire cache ne va charger, au maximum, qu'une seule même ligne de votre matrice. Pour mieux en tirer profit, il est plus intelligent de changer l'ordre des deux boucles internes, afin que la boucle *rapide* se fasse sur les lignes, qui se situent donc dans le cache.

```
double** ikj(double **A, double **B, int m, int n, int l) {
  double **C = zerosMatrix(m, l);
  for (int i = 0; i < m; i++) {
    for (int k = 0; k < n; k++) {   // Hop on echange ces
      for (int j = 0; j < l; j++) { // deux lignes :o
        C[i][j] += A[i][k] * B[k][j];
      }
    }
   }
  return C;
}
```

Encore mieux, on pourrait allouer une très grande colonne (tableau) pour notre matrice, au lieu d'un tableau de tableaux, afin de tirer encore mieux profit d'un éventuel *caching* ainsi qu'en évitant l'*overhead* lié au fait d'avoir un pointeur de pointeurs. Accéder à $A_{i,j}$ se fait via `A[i*m + j]`. C'est d'ailleurs comme ça qu'est construite la matrice du système dans `fem.c`, même si le professeur a ensuite utilisé une petite astuce pour vous permettre de quand même utiliser l'indexation `A[i][j]` :-)

```
double* zerosColMatrix(int m, int n) {
  double *M = calloc(sizeof(double), m * n);
  return M;
}
```

```c
double* ikjCol(double *A, double *B, int m, int n, int l) {
  double *C = zerosColMatrix(m, l);
  for (int i = 0; i < m; i++) {
    for (int k = 0; k < n; k++) {
      for (int j = 0; j < l; j++) {
        C[i*m + j] += A[i*m + k] * B[k*n + j];
      }
    }
   }
  return C;
}
```

Finalement, comme on a eu la flemme d'optimiser notre code, on s'est permis de tester notre programme avec et sans le *flag* `-O3`[1], parce que... Pourquoi pas ? ;-)

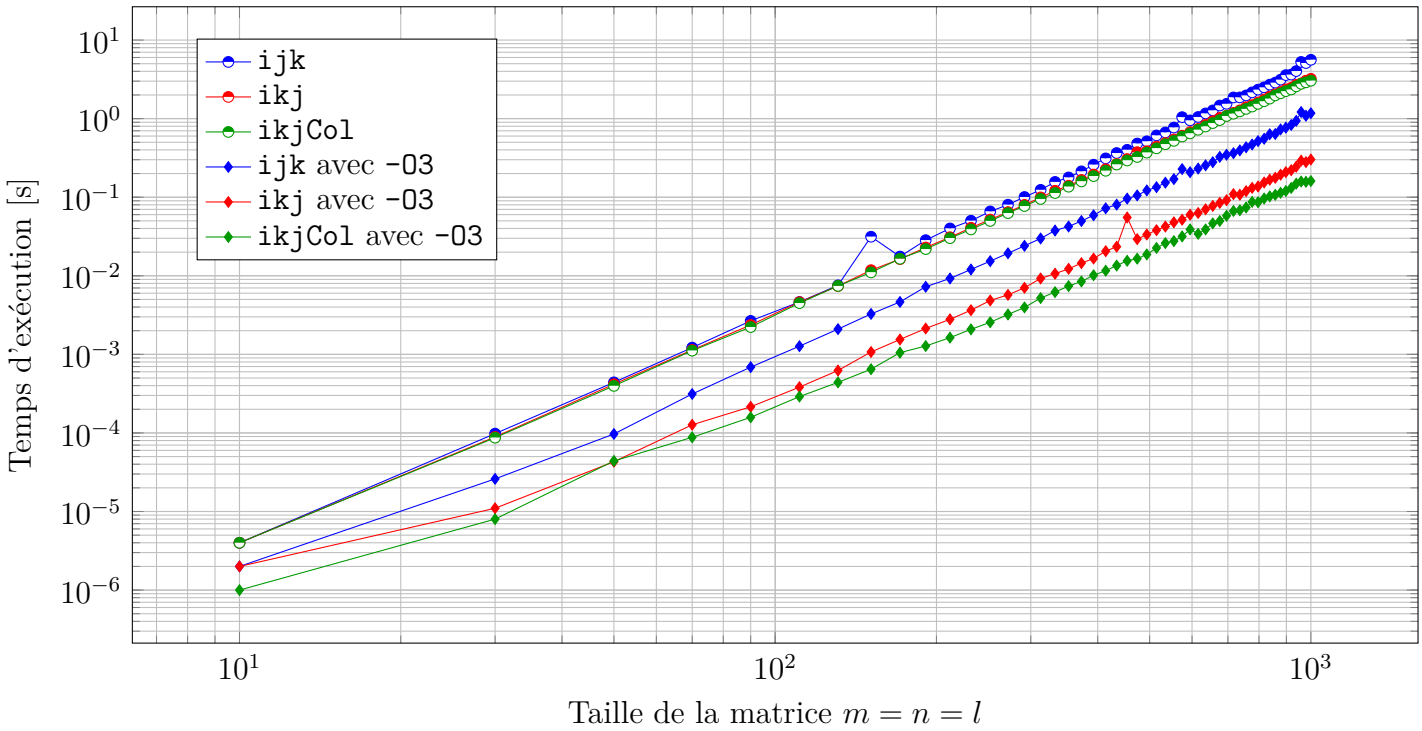


FIGURE 2 – Évolution du temps d'exécution des différentes implémentations en fonction de la taille des matrices.

Sans surprise, la compilation avec `-O3` permet d'accélérer l'exécution du code. Ici, on a un facteur 10, mais cela dépend fortement de votre code et de comment il peut être optimisé par le compilateur. De plus, on observe encore un facteur 10 de rapidité entre la méthode `ijk` et la méthode `ikjCol`. **Morale de l'histoire**, le cache c'est important !

La FIGURE 3 illustre, elle, le hitrate du cache : plus il est élévé, plus la mémoire cache remplit bien son rôle !

Attention : 100 % de hitrate ne signifie pas que tout se trouve dans la mémoire cache, non non non... :-) Les chiffres sont surtout là pour être comparés entre-eux, mais par pour leur valeur absolue.

1. Plus plus d'infos : `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

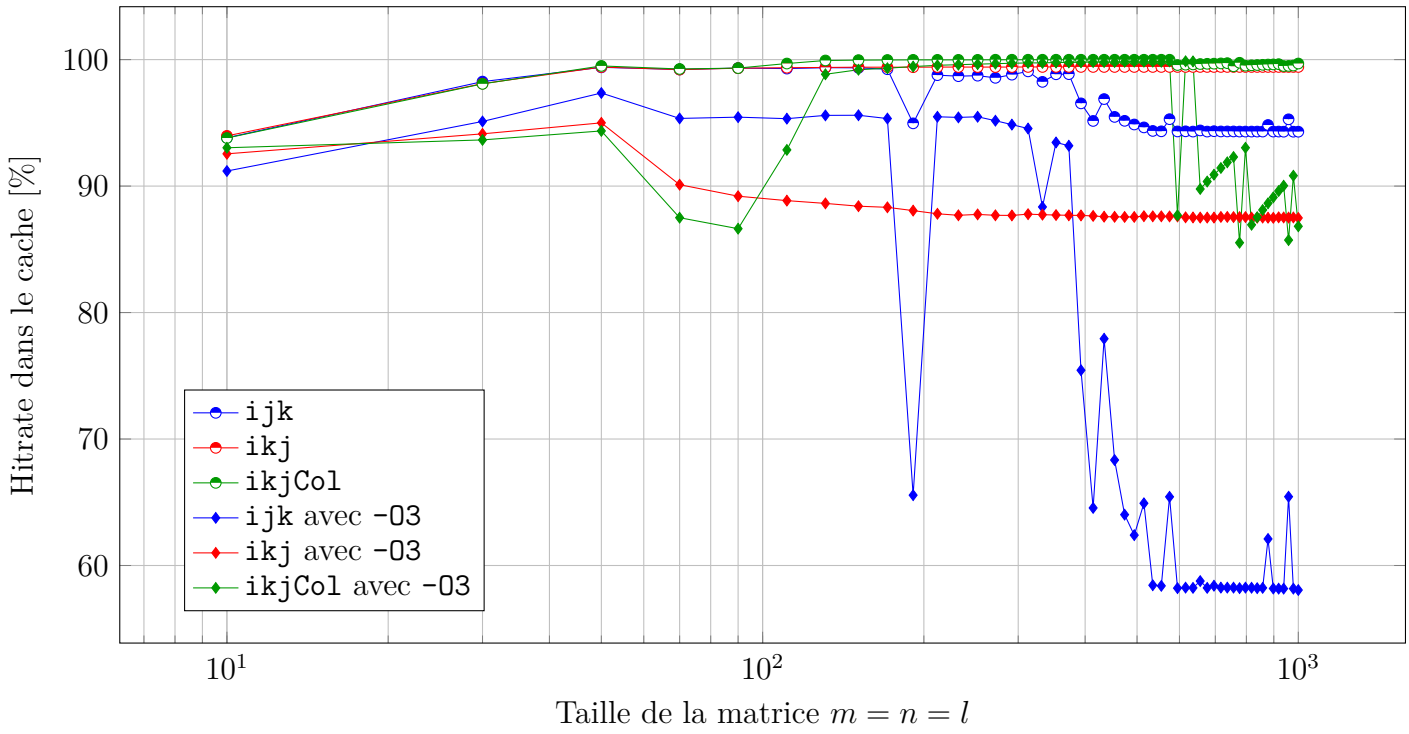


FIGURE 3 – Estimation, via l'outil `perf`, du hitrate moyen des différentes implémentations en fonction de la taille des matrices.

2.1.3 Conclusion

3 Liens avec l'APE 5

Comme on se dit que ça vous intéresse vraiment beaucoup tout ça [2], on propose, avant de passer au devoir, de traverser les différents codes à réaliser ou à comprendre pour la séance 5.

3.1 Exercice 13

Dans cet exercice, on vous a introduit la décomposition d'une matrice symétrique définie positive comme étant le produit d'une matrice triangulaire inférieure et de sa transposée : $\boldsymbol{A} = \boldsymbol{L} \times \boldsymbol{L}^T$. Dans le cas spécifique de cet exercice, la matrice $\boldsymbol{L}$ ne contient que deux diagonales non nulles : $\vec{d}$ et $\vec{e}$.

La résolution de $\boldsymbol{A} \times \boldsymbol{X} = \boldsymbol{B}$, au lieu d'utiliser une éliminiation gaussienne classique en $\mathcal{O}(n^4)$ [3], peut se faire de la manière suivante :

1. Calcul de $\boldsymbol{L}$ via ses 2 diagonales $\sim \mathcal{O}(2n)$
2. Résolution par substition avant de $\boldsymbol{L} \times \boldsymbol{Y} = \boldsymbol{B} \sim \mathcal{O}(n^2)$, avec $\boldsymbol{Y} = \boldsymbol{L}^T \times \boldsymbol{X}$
3. Résolution par substition arrière de $\boldsymbol{L}^T \times \boldsymbol{X} = \boldsymbol{Y} \sim \mathcal{O}(n^2)$

Le tout nous amène à une complexité de $\mathcal{O}(n^2)$, au lieu de $\mathcal{O}(n^4)$, un gain énorme !

```c
void solveCholesky(double **B, int n){
  double *d = malloc(sizeof(double) * n);
  double *e = malloc(sizeof(double) * (n-1));

  // Calcul de ei et di
  d[0] = sqrt(2);
  for(int i = 0; i < n-1; i++){
    e[i] = -1 / d[i];
    d[i+1] = sqrt(2 - e[i]*e[i]);
  } // On aurait aussi pu utiliser la formule explicite :)

  // Resoudre pour y = LT*x = inv(L)*B
  for(int i = 0; i < n; i++){   // Bien remarquer que la boucle sur les j vient
    for(int j = 0; i < n; j++){ // du fait que B est une matrice :)
      if(i != 0){
        B[i][j] -= B[i-1][j] * e[i-1]
      }
      B[i][j] /= d[i];
    }
  }

  // Resoudre pour x = inv(LT)*y
  for(int i = n-1; i >= 0; i--){
    for(int j = 0; i < n; j++){
      if(i != n-1){
        B[i][j] -= B[i+1][j] * e[i]
      }
      B[i][j] /= d[i];
    }
  } // Note : ce code pourrait etre bien mieux optimise
}   // en pre-calculant 1/d (ou en utilisant e), etc.
```

2. C'est pas pour rien que `C` ça rime avec "stylé".
3. Ici, on résout n systèmes en parallèle, chacun prenant $\mathcal{O}(n^3)$ opérations.

Note : Ici, la matrice $\boldsymbol{B}$ va contenir la solution de $\boldsymbol{X}$ après l'appel à notre fonction, c'est ce qu'on appelle une résolution "en place", ce qui permet de sauver de l'espace mémoire. Également, le gros gain en complexité est du au fait que $\boldsymbol{L}$ ne possède que deux diagonales non nulles. En toutes généralités, on peut espérer, pour $\boldsymbol{B}^{n\times n}$, une complexité en $\mathcal{O}(n^3)$ via la décomposition de Cholesky.

3.2 Exercice 15

Le code présenté dans l'exercice 15 a pour objectif d'implémenter la petite astuce permettant l'indexation `A[i][j]` tout en gardant un stockage colonne, comme présenté au point 2.1.2. Ici, on vous détaille chaque étape pour que vous maitrisiez les tableaux `C` comme jamais !

```
femFullSystem *femFullSystemCreate(int size){
  femFullSystem *theSystem = malloc(sizeof(femFullSystem));
  theSystem->A = malloc(sizeof(double*) * size);
  theSystem->B = malloc(sizeof(double) * size * (size+1));

  theSystem->A[0] = theSystem->B + size;
  theSystem->size = size;

  for (int i=1 ; i < size ; i++)
    theSystem->A[i] = theSystem->A[i-1] + size;

  return theSystem;
}

femBandSystem *femBandSystemCreate(int size, int band) {
  femBandSystem *myBandSystem = malloc(sizeof(femBandSystem));
  myBandSystem->B = malloc(sizeof(double)*size*(band+1));
  myBandSystem->A = malloc(sizeof(double*)*size);

  myBandSystem->size = size;
  myBandSystem->band = band;
  myBandSystem->A[0] = myBandSystem->B + size;

  for (int i=1 ; i < size ; i++)
    myBandSystem->A[i] = myBandSystem->A[i-1] + band - 1;

  return(myBandSystem);
}
```

Le but de ce code est d'allouer, en mémoire, les systèmes (complet et bande) que l'on va résoudre. On va d'abord regarder le système complet : on commence par allouer notre structure qui va représenter notre système. Ensuite, on va allouer `A` qui va être un pointeur qui pointe vers un tableau de pointeurs, donc `A` est bien de type `double**` (c'est bien comme ça qu'on déclare un tableau à double entrée).

Ensuite, on s'attaque à `B` qui lui va être un pointeur vers un tableau de `double` de taille `size*(size+1)`. En fait, l'appel à `malloc` est coûteux en temps et il vaut mieux déclarer une fois un tableau de `size*(size+1)`, qui va ensuite être virtuellement découpé en plusieurs, que faire l'inverse. Ici, on va avoir besoin de `size` éléments pour le vecteur $\boldsymbol{B}$ et `size*size` éléments pour la matrice $\boldsymbol{A}$, on préfère donc faire qu'un seul appel à `malloc`.

Sachant cela, il faut encore associer aux éléments de `theSystem->A` une adresse (vu que `A` est un tableau de pointeurs). On va donc utiliser une propriété des pointeurs qui fait que l'on peut appliquer des opérations arithmétiques dessus :

```c
double *myPointer = 1000;

myPointer++; // maintenant myPointer vaut 1008
// + 8 car double est sur 64 bits = 8 octets
```

Pour un entier, si on applique l'opération `++`, on rajoute 1 à l'élément : il va donc, par exemple, passer de 22 à 23. Un pointeur (qui représente une addresse en mémoire) va passer à la prochaine valeur **logique**. Petit rappel : la mémoire est composée de *bits*. La plus petite unité addressable est appelée *byte* et un groupement de 8 *bits* est appelé un *octet* (donc il y a bien une différence entre un *byte* et un *octet*). Toutefois, sur la plupart des ordinateurs, un *byte* vaut extactement un *octet*, ce qui fait qu'on interchange souvent ces deux termes. Une variable va être caractérisée par un type (`int`, `double`...) qui va nous dire comment il faut interpréter la suite de 8 *bits*. Ce type va aussi caractériser le nombre d'*octets* sur lequel est encodé notre variable. Un pointeur connait ce type et donc il va effectuer des incréments de la même taille que le nombre d'*octets* sur lequel est codé la variable.

Tout ça pour dire que, lorsqu'on fait `++` sur un pointeur, ça veut juste dire qu'on pointe vers la prochaine zone après la variable actuelle. Le code pour la partie du système complet devient tout de suite plus clair. Une représentation schématique est faite sur la FIGURE 4.

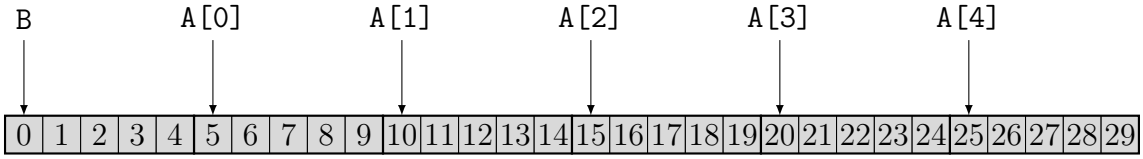


FIGURE 4 – Représentation de la mémoire après le `femFullSystemCreate` pour un système de taille `size = 5`.

Maintenant, on va s'intéresser au système bande dont le but est de réduire l'espace mémoire à une sous partie de notre matrice (et faire moins d'opération dans l'élimination de notre système). Dans le code, on voit bien que notre nombre d'éléments est réduit vu que le `malloc` de `B` n'est plus que de taille `size*(band+1)`. Mais, on souhaite aller encore plus loin et pouvoir utiliser notre vecteur `A` comme dans le système précédent. Normalement, on aurait une matrice comme à la FIGURE 5.

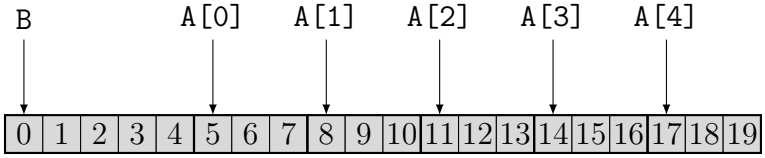


FIGURE 5 – Représentation de la mémoire après le `femFullSystemBand` si on utilise pas les super tricks avec `size = 5` et `band = 3`.

Le principal défaut est que, si on veut accéder à l'élément qui correspond à l'élément `A[3][4]` de notre système complet, on doit réaliser un décalage vers la gauche de notre colonne par notre numéro de ligne actuel et donc accéder à l'élément `A[3][4-3] = A[3][1]`. Votre gentil professeur a donc imaginé une manière (farfelue) de résoudre ce problème. Il va réaliser ce décalage quand il va attribuer aux différents éléments de `A` leur adresse. Vous pouvez remarquer :

— Une manière récursive de définir `A[i]`
— Un facteur `-1`

Ainsi, on va tomber sur le schéma de la FIGURE 6.

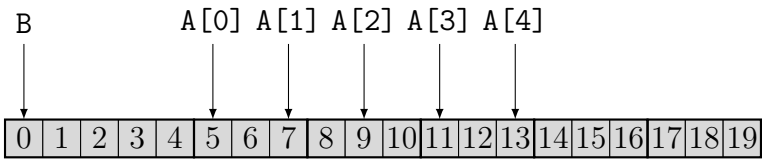


FIGURE 6 – Représentation de la mémoire après le `femFullSystemBand` avec les super tricks avec `size = 5` et `band = 3`.

Ce qui nous permet bien d'accéder de la même manière à notre système bande qu'à notre système complet. On peut retrouver dans la matrice suivante les différents noeuds utilisés :

$$\begin{bmatrix} 5 & 6 & 7 & & \\ & 8 & 9 & 10 & \\ & & 11 & 12 & 13 \\ & & & 14 & 15 \\ & & & & 17 \end{bmatrix} \tag{2}$$

Note : Les éléments *16*, *18* et *19* sont déclarés dans notre matrice mais ne sont jamais utilisés dans les calculs.

3.3 Exercice 16

L'exercice 16, lui, fait appel à des concepts un peu plus *tricky* du langage `C`. Une chose très pratique avec les langages de progammation orientés objet, c'est l'héritage : si une classe parent possède une fonction, toutes les sous-classes de cette dernière en hériteront. Comme le `C` ne possède pas de classe, on peut user d'astuces pour quand même profiter de l'héritage, ce qui évite de devoir écrire plein de fois le même code. C'est, en autres, ce que l'on va voir en analysant le code ci-dessous.

```
typedef struct {
  double *B;
  double **A;
  int size;
}femFullSystem;

typedef struct {
  femSolverType type;
  femFullSystem *local;
  void *solver;
}femSolver;

femSolver *femSolverCreate(int sizeLoc) {
  femSolver *mySolver = malloc(sizeof(femSolver));
  mySolver->local = femFullSystemCreate(sizeLoc);
  return (mySolver);
}

femSolver *femSolverFullCreate(int size, int sizeLoc) {
  femSolver *mySolver = femSolverCreate(sizeLoc);
  mySolver->type = FEM_FULL;
```

```
  mySolver->solver = (femSolver *)femFullSystemCreate(size);
  return(mySolver);
}

femSolver *femSolverBandCreate(int size, int sizeLoc, int band) {
  femSolver *mySolver = femSolverCreate(sizeLoc);
  mySolver->type = FEM_BAND;
  mySolver->solver = (femSolver *)femBandSystemCreate(size,band);
  return(mySolver);
}
```

Voici donc, les réponses que nous proposons aux différentes questions :

1. La structure `femSolver` contient les donnée pour représenter le solveur dans notre mémoire (et une matrice locale)
2. La fonction `femSolverCreate` va créer notre structure `femSolver` en mémoire et aussi la matrice locale en mémoire
3. Le pointeur de type `void*` est un pointeur générique (rappel du devoir *Edges*). Pourquoi a-t-on besoin de ce pointeur générique ? Car on veut pouvoir stocker dans la structure `femSolver` un pointeur de type `femSolver` et que, lorsqu'on déclare la structure, elle n'existe pas encore donc elle ne sait pas ce qu'est un `femSolver`. On utilise donc le pointeur générique `void*` à la place. Une manière de contourner cela est de déclarer à l'avance qu'une structure `femSolver` va exister :

   ```
   typedef struct femSolver femSolver;

   struct femSolver {
     femSolverType type;
     femFullSystem *local;
     femSolver *solver;
   };
   ```

   La première ligne indique que l'on définit une structure `struct femSolver` nommée `femSolver` ainsi, lorsqu'on va définir le contenu même de la structure, le compilateur sait qu'il existe une structure nommée `femSolver`
4. Si on regarde la documentation, on voit que `femFullSystemCreate` renvoie un pointeur de type `femFullSystem` alors que, nous, on veut travailler avec des pointeurs de type `femSolver`. On va donc convertir notre pointeur qui pointe d'un `femFullSystem` vers un pointeur `femSolver`, c'est ce qu'on appel un typecast : le changement de type d'un élément vers un autre. Note : écrire ceci

   ```
   float myPotatoes = 100.0;
   ```

   pour déclarer un float va effectuer un typecast d'une variable `double` vers `float` tout comme on peut convertir un `int` en `double`. Effectuer un typecast est une opération coûteuse, il ne faut donc l'effectuer que si on en a besoin et, donc, la bonne manière de déclarer un `float` est la suivante :

   ```
   float myPotatoes = 100.0f;
   ```
5. C'est une implémentation de l'héritage (*Inheritance*)
6. Le `C++` est un language orienté objet : on utiliserait donc une classe générale `solver`. Chacun des solveurs (`FullSystemSolver`, `BandSolver`, `IterativeSolver`) implémentés devrait hériter de cette classe. Ainsi, on pourrait utiliser de manière générale chacun des 3 solveurs sans se soucier du type de solveur `mySolver.solve()`

4 Résolution du problème

Voilà arrivé le moment tant attendu : la correction du devoir ! Cette semaine encore, bien comprendre la matière des TPs permet de plus facilement résoudre le devoir, c'est d'ailleurs pour cela qu'on a tenu à d'abord commencer par cela ;-)

4.1 `femMeshRenumber`

Dans cette fonction, le but est de renuméroter votre maillage afin de réduire la taille de bande. Comme cela a été vu en TP, toutes les numérotations ne se valent pas : l'une peut mener à une bande plus petite qu'une autre, mais les résultats dépendent aussi fortement du maillage utilisé.

Pour garder un code simple, on vous demande de rajouter deux numérotations : une selon les coordonnées X, et une seconde selon les Y. Pour ce faire, on peut utiliser la désormais connue fonction `qsort` pour trier les noeuds selon nos coordonnées, et on va ensuite utiliser le résultat du tri pour renuméroter les éléments, d'où le tableau `inverse`.

```
void femMeshRenumber(femMesh *theMesh, femRenumType renumType) {
  int i;
  int *inverse = (int*) malloc(sizeof(int)*theMesh->nNode);

  for (i = 0; i < theMesh->nNode; i++)
    inverse[i] = i;

  switch (renumType) {
    case FEM_NO :
      break;
    case FEM_XNUM :
      theGlobalArrayPos = theMesh->X;
      qsort(inverse, theMesh->nNode, sizeof(int), compareNodePos);
      break;
    case FEM_YNUM :
      theGlobalArrayPos = theMesh->Y;
      qsort(inverse, theMesh->nNode, sizeof(int), compareNodePos);
      break;
    default : Error("Unexpected renumbering option");
  }

  for (i = 0; i < theMesh->nNode; i++)
    theMesh->number[inverse[i]] = i;
  free(inverse);
}
```

Comme pour le précédent devoir, il vous faut définir la fonction qui servira à comparer vos noeuds selon soit leur abscisse, soit leur ordonnée : ici, `theGlobalArrayPos` va pointer soit vers `X`, soit vers `Y`.

```
int compareNodePos(const void *nodeOne, const void *nodeTwo) {
  int *iOne = (int *)nodeOne;
  int *iTwo = (int *)nodeTwo;
  double diff = theGlobalArrayPos[*iOne] - theGlobalArrayPos[*iTwo];
  return (diff < 0) - (diff > 0); // Retourne -1 / 0 / 1
}
```

4.2 femMeshComputeBand

Pour un système creux, on définit la largeur de bande comme étant la plus grande distance entre le noeud minimal et le noeud maximal d'une même ligne de notre matrice. La largeur de bande va directement impacter le nombre de colonnes que notre élimination gaussienne va devoir parcourir.

Pour cette fonction, on pourrait construire la matrice de notre système complet et, ensuite, regarder sur cette matrice la largeur de bande. Mais cette méthode est lente et nous force à allouer une grande matrice en mémoire. On va donc préférer parcourir les élements de notre maillage qui détiennent les informations nécessaire pour calculer notre largeur de bande.

```
#define MAX(a,b) ((a>b)?(a):(b)) // Pour comprendre les macros
#define MIN(a,b) ((a<b)?(a):(b)) // cf. notre precedent solutionnaire

int femMeshComputeBand(femMesh *theMesh) {

  int myMax, myMin, myBand, map[4];
  int nLocal = theMesh->nLocalNode;
  myBand = 0;

  for(int iElem = 0; iElem < theMesh->nElem; iElem++) {
    for (int j = 0; j < nLocal; ++j)
      map[j] = theMesh->number[theMesh->elem[iElem*nLocal+j]];

    // On trouve le noeud maximum et minimum
    myMin = map[0];
    myMax = map[0];
    for (int j = 1; j < nLocal; j++) {
      myMax = MAX(map[j],myMax);
      myMin = MIN(map[j],myMin);
    }

    if (myBand < (myMax - myMin))
      myBand = myMax - myMin;
  }

  return(++myBand); // On incremente de 1 myBand avant de le renvoyer (formule)
}
```

4.3 `femBandSystemAssemble`

Ici, on doit assembler la matrice `A`. Comme précisé dans les consignes, ce n'est pas très compliqué grâce au *super trick* du prof. pour accéder à notre matrice bande comme on le fait avec la matrice du système complet. Il faut juste faire attention à ne pas sortir de la bande (FIGURE 7).

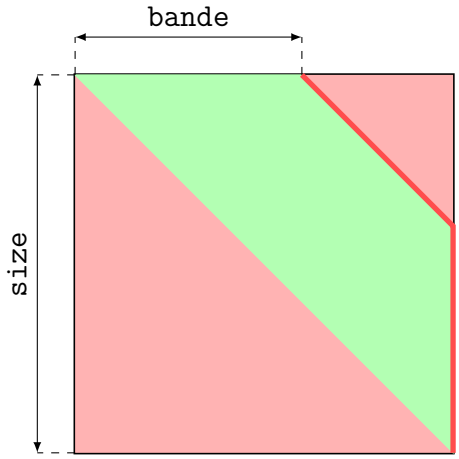


FIGURE 7 – Représentation des zones de travail et à éviter.

On est certain de ne jamais dépasser dans la zone rouge supérieure vu que ça voudrait dire que notre calcul de largeur de bande est mauvais et qu'il existe une contribution de la part d'un autre noeud. On doit juste s'assurer de ne pas remplir la partie inférieur de notre matrice (du à la symétrie de cette dernière). Pour cela, il suffit donc de rajouter une comparaison entre le numéro de ligne et de colonne.

```
void femBandSystemAssemble(femBandSystem* myBandSystem, double *Aloc, double *Bloc,
    int *map, int nLoc) {
  for (int i = 0; i < nLoc; i++) {
    int myRow = map[i];

    for(int j = 0; j < nLoc; j++) {
      int myCol = map[j];

      // Ici, on ne regarde que les elements superieurs de la diagonale
      if (myCol >= myRow) {
        myBandSystem->A[myRow][myCol] += Aloc[i*nLoc+j];
      }
    }
    myBandSystem->B[myRow] += Bloc[i];
  }
}
```

4.4 femBandSystemEliminate

Et maintenant, il ne faut plus qu'éliminer notre matrice bande pour résoudre notre système. Ici encore, il ne faut pas sortir de la bande (FIGURE 7) :

— Pour le calcul du facteur lors de l'élimination gaussienne, normalement, on récupère les élements sous la diagonale. Mais, ici, c'est une zone interdite car on a profité de la symétrie de notre matrice pour ne pas les stocker. Donc, pour le calcul du facteur, on va utiliser l'élément symétrique qui se trouve dans la partie supérieure.

— Pour l'élimination gaussienne, on applique la soustraction entre 2 lignes de la matrice avec un certain facteur. Ici, on doit bien faire attention à n'appliquer la soustraction que pour les éléments de la bande. Pour ce faire, on va calculer le terme `jend` qui va décrire la ligne rouge de la FIGURE 7 et s'arrêter au niveau de ce terme.

```
double  *femBandSystemEliminate(femBandSystem *myBand) {
  double  **A, *B, factor;
  int     i, j, k, jend, size, band;
  A    = myBand->A;
  B    = myBand->B;
  size = myBand->size;
  band = myBand->band;

  /* Gaussian elimination */
  for (k=0; k < size; k++) {
    if ( fabs(A[k][k]) <= 1e-4 ) { Error("Cannot eleminate with such a pivot"); }

    // Limite de la ligne rouge
    jend = MIN(k + band,size);
    for (i = k+1 ; i <  jend; i++) {
      // On recupere l'element symetrique
      factor = A[k][i] / A[k][k];
      for (j = i ; j < jend; j++)
        A[i][j] = A[i][j] - A[k][j] * factor;
      B[i] = B[i] - B[k] * factor;
    }
  }

  /* Back-substitution */
  for (i = (size-1); i >= 0 ; i--) {
    factor = 0;

    // Limite de la ligne rouge
    jend = MIN(i + band,size);
    for (j = i+1 ; j < jend; j++)
      factor += A[i][j] * B[j];
    B[i] = ( B[i] - factor)/A[i][i];
  }

  return(myBand->B);
}
```

Globalement, le code était assez simple car il suffisait de s'inspirer de ce qui était déjà fait pour le système complet dans `fem.c` et d'en changer quelques lignes pour l'adapter au cas d'un système bande.