© *Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>*

L'utilitaire safemira_sparse

Thème - 1 *Motivations*

Les méthodes directes pour matrices creuses, en particulier la factorisation LU ou Cholesky, se déroulent essentiellement en deux phases :

— Une phase dite de *factorisation symbolique*, visant à prédire la structure (creuse) des facteurs L et U afin d'allouer la juste quantité d'espace mémoire pour leur stockage (comme matrice creuse).

— Une phase dite de *factorisation numérique*, où la factorisation est effectivement réalisée, en utilisant la structure et l'espace mémoire définis à la phase précédente.

Un avantage majeur de ces deux phases (plus particulièrement la première) est cette possibilité de décider du choix de structure de données pour le stockage des facteurs L et U. En effet puisqu'on connaît les positions des entrées non nulles dans ces matrices, on peut (et c'est ce qui est fait en pratique) utiliser des structures de données statiques (permettant des accès et insertions en temps constants) comme des tableaux pour représenter ces matrices. Les formats de stockage *compressed sparse rows (CSR)* et *compressed sparse columns (CSC)* sont des exemples de structures utilisant uniquement des tableaux pour stocker les matrices creuses.

Malheureusement la factorisation symbolique, pour être efficace, fait usage des outils assez sophistiqués comme la théorie des graphes (arbres d'élimination) pour prédire les structures des facteurs L et U.

Puisque nous n'envisageons pas dans ce cours de faire appels à ces notions de graphes. Nous optons pour une structure de données dynamique qui permet de jumeler les deux phases de factorisation en une seule. C'est le but de l'utilitaire que nous décrivons dans les lignes ci-dessous.

Thème - 2 *Description*

L'utilitaire `safemira_sparse` stocke une matrice creuse comme un tableau de lignes, chaque ligne étant une liste chaînée.

Ainsi la structure définissant une matrice creuse est donnée par :

Listing 1 – Représentation d'une matrice creuse

```
typedef struct SparseMatrix
{
  int n;  // nombre de lignes
  int m;  // nombre de colonnes
  pSparseElement *v;
} SparseMatrix, *pSparseMatrix;
```

La structure définissant une ligne est quant-à-elle donnée par :

Listing 2 – Représentation d'une ligne de la matrice

```
typedef struct SparseElement
{
  int    col;                  // numero de la colonne
  double val;                  // valeur stockee
  struct SparseElement *m_next; // element suivant dans la ligne
} SparseElement, *pSparseElement;
```

C'est en terme de cette structure de données que les fonctions permettant de manipuler la matrice matrice creuse sont définies . Citons quelques unes de ces fonctions.

Listing 3 – Fonctions permettant de travailler avec la matrice creuse

```
SparseMatrix* sm_create (int n, int m);
void sm_free (void *p);
void sm_set_entry (SparseMatrix * sm, int i, int j, double val);
void sm_insert_value (SparseMatrix * sm, int i, int j, double val);
double sm_get_value (const SparseMatrix * sm, int i, int j);
int sm_nnz (const SparseMatrix * sm);// nombre d'elements non nuls
void sm_print (FILE * fid, const SparseMatrix * sm);
void sm_spy_gnuplot(const SparseMatrix * sm, const char* filename);
//factorisation lu
void sm_lu(SparseMatrix * sm);
void sm_write_metis_graph(const SparseMatrix * sm, const char* filename);
void sm_read_metis_graph(const SparseMatrix * sm, int* pinv,
                         const char* filename);
//permutation avec l'utilitaire Metis
SparseMatrix* sm_metis_reorder(const SparseMatrix * sm, int* ip);
SparseMatrix* sm_duplicate(const SparseMatrix * smfrom); //copie
SparseMatrix* sm_transpose(const SparseMatrix * smfrom);
//z = sm1 + b * sm2
SparseMatrix* sm_add(const SparseMatrix * sm1, double b, const SparseMatrix *sm2);
// y = A*x + a*y
void sm_gaxpy(const SparseMatrix * sm, const double* x, int m, double a, double*y, int n);
//resolution avec partie triangulaire inferieure. On suppose des 1 sur la diagonale
void sm_solve_lower (const SparseMatrix* sm, double *v);
//resolution avec la partie triangulaire superieure
void sm_solve_dupper (const SparseMatrix* sm, double *v);
```

Remark 0.0.1.

`safemira_sparse` offre aussi une structure de données permettant de gérer les vecteurs creux (voir Listing 4).
Cette structure était initialement prévue pour représenter une ligne de la matrice creuse.
C'est un exercice intéressant de compléter cette structure afin de réaliser toutes les opérations d'algèbre linéaire sur les vecteurs.

Listing 4 – Représentation d'un vecteur creux

```
typedef struct SparseVector
{
  SparseElement *root;
} SparseVector, *pSparseVector;
pSparseVector sv_create ();
void   sv_free (void *v);
int    sv_size (const pSparseVector v);
int    sv_nnz (const pSparseVector v);
void   sv_print (FILE * fid, const SparseVector * pv, char *separator);
void   sv_add_value (SparseVector * v, int j, double val);
double sv_get_value (const SparseVector * v, int j);
void   sv_axpy(SparseVector * y, double a, const SparseVector * x, int startIdx);
```

Thème - 3 *Illustrations*

L' utilitaire `safemira_sparse` a été conçu exclusivement pour le présent cours. Il est loin d'être complet et ne doit pas être utilisé pour les codes de production.

L'utilitaire est en encore en chantier, et par conséquent en constante ajustement : les noms des fonctions et variables sont appelées à évoluer jusqu'à obtention d'une version stable de l'utilitaire.

Néanmoins cet utilitaire permet de réaliser l'objectif pédagogique pour lequel il a été conçu : *illustrer la mise en oeuvre de factorisation LU d'une matrice creuse et mettre en évidence le phénomène de remplissage ainsi que les conséquences de la renumérotation.*

L'illustration ci-dessous (voir Listing5) indique les fonctions principales à appeler.

Listing 5 – Illustration

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mcheck.h>

#include "safemira_sparse.h"
#include "cpu_timer.h"

void
test_SparseMatrix_LU ()
{
  const int n = 30; //taille des matrices
  SparseMatrix *sm = sm_create (n, n); // creation d'une matrice
  double x[n], b[n], z[n] ;
  // diagonale
  for (int i = 0; i < n; ++i)
    {
      sm_insert_value (sm, i,i,2.0);
      x[i] = (double) i;        // x stocke temporairement la solution exacte
    }
 //sous/sur diagonale
  for (int i = 0; i < n - 1; ++i)
    {
      sm_insert_value (sm, i, i + 1, -1.0);
      sm_insert_value (sm, i + 1, i, -1.0);
    }
  // creation du second membre du systeme b = sm * x
  sm_gaxpy(sm, x,n, 0, b,n);
  sm_spy_gnuplot(sm,"sm.gnu"); //affichage au format gnuplot
  int* ip = (int*) calloc(sm->n, sizeof(int));
  //creation d'une matrice renumererotee par bissection emboitee
  SparseMatrix *smR = sm_metis_reorder(sm, ip);
  //factorisation de la matrice initiale
  sm_lu(sm);
  sm_spy_gnuplot(sm,"smlu.gnu"); //affichage au format gnuplot
  sm_spy_gnuplot(smR,"smR.gnu");   //affichage au format gnuplot
  sm_lu(smR);                     //factorisation de la matrice renumerotee
  sm_spy_gnuplot(smR,"smRlu.gnu"); //affichage de la matrice factorisee
  // Resolution du systeme lineaire avec la matrice factorisee
  for(int i =0; i< n; ++i) z[ip[i]] = b[i];
  sm_solve_lower(smR,z);
  sm_solve_dupper(smR,z);
  for(int i =0; i< n; ++i)  b[i] = z[ip[i]];// b a la solution finale
  double error = 0;
  for(int i =0; i< n; ++i) error += fabs(x[i]-b[i]);
  printf("|| x - xe ||_L1 = %f \n", error);
  //liberation des ressources
  sm_free(sm);
```

```
  sm_free(smR);
  free(ip);
}
int
main (int argc, char **argv)
{
  mtrace();
  test_SparseMatrix_LU();
  muntrace();
  return 0;
}
```

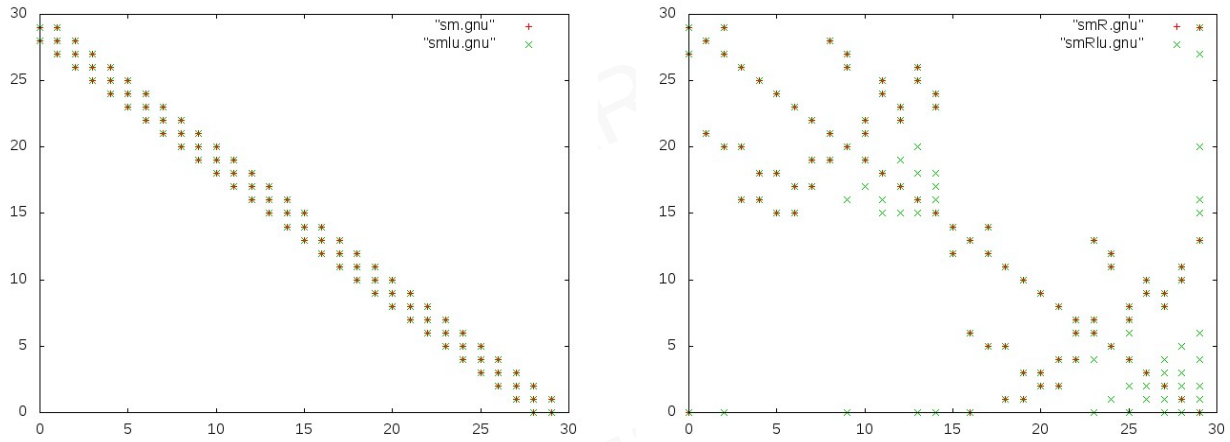

FIGURE 1 – Figure obtenue à la suite du Listing5. Il faut s'y référer pour les légendes. A gauche : la matrice de départ et sa factorisation. A droite : la matrice renumérotée et sa factorisation. La renumérotation a été réalisée par la méthode de bissection emboîtée grâce l'outil METIS. On observe un léger remplissage dans la factorisation de la matrice renumérotée (la matrice de départ avait déjà une structure empêchant le phénomène de remplissage !).