TT OCS Compte rendu

Constanza Corentin

Novembre 2021

1 Introduction

On définit le concept de structure Ligne qui regroupe au sein d'une même entité informatique trois attributs pour pouvoir représenter un coefficient d'une ligne d'une matrice.

— un **double** contenant le coefficient

— un **int** contenant l'indice de colonne

— un **Ligne*** pointant sur le coefficient suivant dans la Ligne ou NULL s'il n'existe pas.

Cette structure Ligne permet de construire une liste simplement chaînée pour représenter tous les coefficients d'une ligne d'une matrice où chacun de ses éléments stocke la valeur et l'indice de colonne d'un coefficient non nul et un pointeur donnant l'adresse de l'élément suivant. La ligne se termine lorsque le pointeur ptsuiv est NULL. Pour éviter de devoir déclarer une variable Ligne avec **struct Ligne** var, on définit le type **typeLigne** permettant de déclarer **typeLigne** var.

```
typedef struct Ligne {
        double A;
        int C;
        struct Ligne *ptsuiv;
} typeLigne;
```

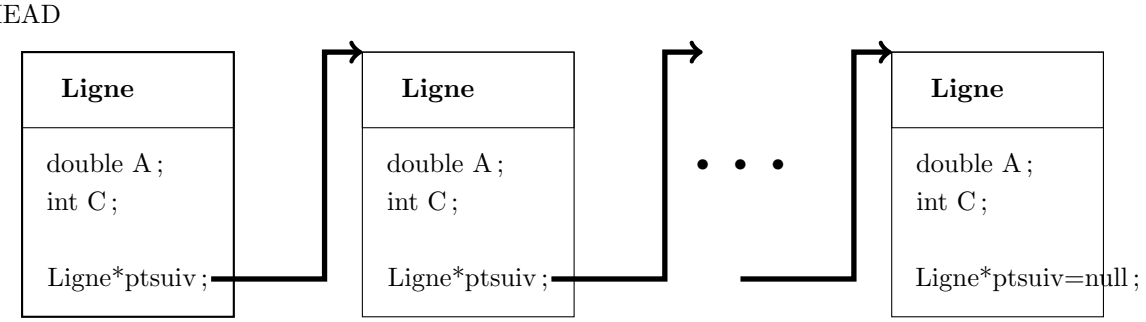
Cette structure dynamique permet de stocker dans un tableau de typeLigne de taille n les matrices creuses comme par exemple la matrice $A \in \mathbb{R}^{9\times 9}$ suivante

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \quad (1)$$

Même si une ligne n'a que des coefficients nuls, il y a quand même création d'un élément pour cette ligne. On prendra comme convention que si l'indice de colonne de ce premier élément est -1 alors la ligne a tous ces coefficients à 0. A leur création, on initialisera ces lignes qui n'ont pas de coefficients encore affectés, en mettant l'indice de colonne du premier élément à -1 :

```
typeLigne *MatA;
MatA=(typeLigne*) malloc(9*sizeof(typeLigne));
for (i=0; i<9; i++){MatA[i].C=-1;MatA[i].A=0.; MatA[i].ptsuiv=NULL;}
```

Une fois rempli, un élément de MatA[i] représentant la ligne i possède la structure suivante :



On **cherchera à ranger** dans l'ordre croissant des indices de colonnes les coefficients dans la ligne.

2 Fonctions auxiliaire :

Afin de pouvoir effectuer les différents tests il a fallu créer deux fonctions auxiliaires pour gérer l'affichage de notre matrice, la libération de mémoire de notre liste chainé et d'une fonction d'initialiser nos listes chainées :

```
void freeLigne(typeLigne *l) {
    typeLigne *ptr, *tmp;
    ptr = l->ptsuiv;

    while(ptr != NULL) {
        tmp = ptr->ptsuiv;
        free(ptr);
        ptr = tmp;
    }
}
```

Cette fonction parcours notre liste chainé et libère les cellules une à une. On stock le pointeur suivant dasn un tampon pour ne pas "perdre le fil" (car on libère aussi le pointeur ptsuivant lors du free).

```
void printMat(typeLigne *l, int n) {
    int i;
    int j;
    typeLigne *ptc;
    for (i=0;i<n;i++)
    {
        ptc=&l[i];
        for (j=0;j<n;j++)
        {
            if ((ptc != NULL)&&(ptc->C == j))
            {
                printf("%f",ptc->A);
                ptc=ptc->ptsuiv;
            }
            else
            {
                printf("%f",0.);
            }
        }
        printf("\n");
    }
}
```

Cette fonction prend en entré la matrice que l'on veut afficher et son nombre de ligne, comme ce dernier est connu, on peut utiliser une boucle for. On parcours chaque ligne cellule par cellule. Si l'indice de la colone $j = C$ avec C l'indice de colone du coefficient sur lequel pointe ptc, alors on affiche le coefficient et on passe à la cellule suivante. Sinon on affiche un 0. Cela nous permet d'afficher notre matrice comme si on l'avait stocké normalement dans un tableau 2D ce qui est pratique pour effectuer nos differents test.

```
void initLigne(typeLigne *l) {
    l->C = -1;
    l->ptsuiv = NULL;
}
```

La fonction initLigne est très simple, elle initialise simplement notres liste chainée et à pour effet de reserver la mémoire nécessaire à notre matrice.

3 La fonction insert :

La fonction insert est decomposer en deux parties, la première est une boucle while qui nous permet d'arriver jusqu'a l'emplacement de la cellule que l'on veut insérer (ou modidfier).

```
void insert(typeLigne *MatA, int i, int j, double Aij)
{
```

```
typeLigne *ptc; //pointeur pour acceder a Mat[i]
typeLigne *ptcprec; //pointeur qui sauvegarde l'adresse de la cellule
   precedente
ptc=&MatA[i];
ptcprec=NULL;
if (Aij == 0.0) //on ne stock pas les 0;
    {
            return;
    }
//on parcourt la liste chainee avec une boucle while
while ( (ptc!=NULL) && (ptc->C !=-1) && (ptc->C <j) )
{
    ptcprec=ptc;
    ptc=ptc->ptsuiv;
}
```

La deuxième partie quand à elle devra gérer 5 cas different :

```
//------Test des different cas-----//
```

— Cas 1 : j > C : on ajoute une cellule à la fin de la liste chainée

```
if (ptc==NULL)                  //Cas 1
{
    ptcprec->ptsuiv=(typeLigne*)malloc(sizeof(typeLigne));
    ptcprec->ptsuiv->C=j;
    ptcprec->ptsuiv->A=Aij;
    ptcprec->ptsuiv->ptsuiv=NULL;
}
```

— Cas 2 : C1=-1 : on remplit simplement la première cellule de la liste chainé

```
else if (ptc->C == -1)              //Cas 2
{
    ptc->C=j;
    ptc->A=Aij;
}
```

— Cas 3 : j < C1 : on ajoute une cellule au début de la liste chainée

```
else if ((ptc->C > j)&&(ptcprec == NULL))    //Cas 3
{
    ptcprec=ptc;
    ptc=(typeLigne*)malloc(sizeof(typeLigne));
    ptc->C = ptcprec->C;
    ptc->A = ptcprec->A;
    ptc->ptsuiv = ptcprec->ptsuiv;
    ptcprec->C=j;
    ptcprec->A=Aij;
    ptcprec->ptsuiv=ptc;
}
```

— Cas 4 : j=Ck : on remplace simplement la valeur de A dans la cellule k

```
else if (ptc->C ==j)            //Cas 4
{
    ptc->A = Aij;
}
```

— Cas 5 : Ck < j < Ck+1 : on crée une nouvelle cellule que l'on insert entre les cellules k et k+1

```
else if ((ptc->C >j)&&(ptcprec->C < j))    //Cas 5
{
    ptc=ptcprec->ptsuiv;
    ptcprec->ptsuiv = (typeLigne*) malloc(sizeof(typeLigne));
    ptcprec->ptsuiv->C=j;
```

```
            ptcprec->ptsuiv->A=Aij;
            ptcprec->ptsuiv->ptsuiv=ptc;
        }
```

Nous allons maintenant tester cette fonction : Tout d'abord initialisons notre matrice ainsi que nos variables locales

```
printf("Test de la fonction insert : \n");
int N =3;
int i;
typeLigne a[N];

for (i = 0; i < 3; i++)
{
        initLigne(&a[i]);
}
```

Insérons maintenant nos differents coéfficients

```
//test des cas 1,2 et 3
    insert(&a[0],0,0,2.0);insert(&a[0],0,1,0.0);insert(&a[0],0,2,4.0);
    insert(&a[0],1,0,4.0);insert(&a[0],1,1,6.0);insert(&a[0],1,2,0.0);
    insert(&a[0],2,0,8.0);insert(&a[0],2,1,0.0);insert(&a[0],2,2,9.0);
```

Affichons maintenant cette matrice

```
printf("Matrice A : \n");
printMat(a,N);
printf("\n");
```

Nous obtenons la sortie :

```
Test de la fonction insert :
Matrice A :
( 2.000000 0.000000 4.000000 )
( 4.000000 6.000000 0.000000 )
( 8.000000 0.000000 9.000000 )
```

Ce qui est bien la sortie attendu. Lors de ce test nous avons verifier que les cas 1,2 et 3 était bien implémentés. Nous allons maintenant essayer de remplacer des coefficients déjà existants et d'en inserer un nouveau à la place de l'un des 0 pour verifier les cas 4 et 5. Pour ce faire nous allons mettre des 1 sur la première ligne de notre matrices :

```
//test des cas 4 et 5
    insert(&a[0],0,0,1.0);insert(&a[0],0,1,1.0);insert(&a[0],0,2,1.0);
    printf("Matrice A : \n");
    printMat(a,N);
```

Nous obtenons la sortie :

```
Matrice A :
( 1.000000 1.000000 1.000000 )
( 4.000000 6.000000 0.000000 )
( 8.000000 0.000000 9.000000 )
```

Ce qui est encore une fois le resultat attendu. Notre fonction insert est donc bien implementée.

Enfin on libère la mémoire :

```
for(i=0; i<N; i++)
            {
                    freeLigne(&a[i]);
            }
```

4 Fonction MatveCSR

Nous voulons une fonction qui effectue le produit matrice vecteur $y = Ax$ ou $A \in \mathbb{R}^{N \times N}$. La fonction sera definie comme ci-dessous :

```
void matveCSR(typeLigne *A, double *x, double *y, int n) {
```

Commencons par definir nos variable locales.

```
    int i;
    typeLigne *ptrMat;
    double coeff;
```

Pour effectuer ce produit on utilise une boucle for (car on connait le nombre de ligne N passé en paramètre) et mettre pour chaque ligne l'adresse de celle ci dans notre pointeur que l'on vient de créer. On remet aussi coeff à 0 par définition du produit matrice vecteur. On calcule $y[i]$ avec une boucle while parcourant chaque ligne et faisant la somme des coéfficients de la ligne i.

```
    for(i=0; i<n; i++) {
        coeff = 0.0;
        ptrMat = &A[i];

        while(ptrMat != NULL) {
            coeff += ptrMat->A * x[ptrMat->C];
            ptrMat = ptrMat->ptsuiv;
        }
        y[i] = coeff;
    }
}
```

Nous allons à present tester notre fonction.

On va commencé par créer une matrice ainsi que deux vecteurs. On initialise la matrice et on remplit ensuite les vecteurs.

```
    printf("Test␣produit␣matrice␣vecteur␣:␣\n\n");
        int i;
        int N =3;
    typeLigne A[N];
    double x[N], y[N];
        for(i = 0; i < N; i++)
        {
        initLigne(&A[i]);
    }
```

x est le vecteur que l'ont va multiplier. On lui met donc des 1 pour que le calcul soit simple et une 0 pour montrer que la multiplication par 0 fonction bien. y est le vecteur qui contiendra le résultat. Remplissons maintenant notre vecteur x et notre matrice A que nous afficherons :

```
    insert(&A[0],0,0,3.0);insert(&A[0],0,1,0.0);insert(&A[0],0,2,3.0);
    insert(&A[0],1,0,0.0);insert(&A[0],1,1,2.0);insert(&A[0],1,2,2.0);
    insert(&A[0],2,0,1.0);insert(&A[0],2,1,0.0);insert(&A[0],2,2,4.0);

        printf("Matrice␣A␣:␣\n");
        printMat(A,N);

    for(i = 0; i < N; i++)
        {
                x[i] = 1.0;      //  x = (1, 1, 1)

                y[i] = 0.0;      //init de y
        }
        x[1] = 0.0;          // x = (1 ,0 ,1)
        printf("Vecteur␣x␣:␣\n");
        for(i = 0; i < N; i++)
        {
                printf("(␣%f␣)\n", x[i]);
        }
```

On effectue ensuite le produit matrice vecteur et on affiche y :

```
    matveCSR(&A[0], x, y, 3);
    matveCSR(&A[0], x, y, N);

        printf("Produit␣y␣=␣A␣*␣x␣:␣\n");

    for(i = 0; i < N; i++)
        {
                printf("(␣%f␣)␣\n", y[i]);
        }

    for(i = 0; i < N; i++)
        {
        freeLigne(&A[i]); //liberation de la memoire
    }

        printf("\n");
}
```

On a donc la sortie :

```
Test produit matrice vecteur :

Matrice A :
( 3.000000 0.000000 3.000000 )
( 0.000000 2.000000 2.000000 )
( 1.000000 0.000000 4.000000 )
Vecteur x :
( 1.000000 )
( 0.000000 )
(1.000000 )
Produit y = A * x :
( 6.000000 )
( 2.000000 )
( 5.000000 )
```

Ce qui correspond bien au résultat attendu.

5 Question 3 :

1. **Montrer que** $\alpha_n = \langle r_n, d_n\rangle / \langle d_n, Ad_n\rangle$ et que $r_{n+1} = r_n - \alpha_n A d_n$
   On cherche α tel que $x_{n+1} = x_n + \alpha_n d_n$ minimise $J(x)$ sur $x_0 + K_{n+1} = \{d_0 ... d_n\}$
   On a :

$$J(x) = \tfrac{1}{2}\langle Ax, x\rangle - \langle b, x\rangle$$

   Avec comme condition sur les directions de descent :

$$\langle Ad_i, d_i\rangle / = \begin{cases} = 0 & \text{si } i \neq j \\ \neq 0 & \text{sinon.} \end{cases}$$

   tq on minimise J sur $\{d_0, ..., d_n$. On a donc :

$$\tfrac{\partial J}{\partial \alpha_n}(x_{n+1}) = 0 \text{ or } J(x_{n+1}) = \tfrac{1}{2}\langle Ax_{n+1}, x_{n+1}\rangle - \langle b_{n+1}, x_{n+1}\rangle$$

   D'où
   $J(x_{n+1}) = \frac{1}{2}\langle Ax_n + A\alpha_n d_n, x_n + \alpha_n d_n\rangle - \langle b, x_n + \alpha_n d_n\rangle$
   $= \frac{1}{2}(\langle Ax_n, x_n\rangle + \langle Ax_n, \alpha_n d_n\rangle + \langle A\alpha_n d_n, x_n\rangle + \langle A\alpha_n d_n, \alpha_n d_n\rangle) - \langle b, x_n\rangle - \langle b, \alpha_n d_n\rangle$
   $= \frac{1}{2}(, \langle Ax_n, x_n\rangle + \alpha_n(, \langle Ax_n, d_n\rangle + \langle Ad_n, x_n\rangle,) + \alpha_n^2\langle Ad_n, d_n\rangle,) - \langle b, x_n\rangle - \langle b, \alpha_n d_n\rangle$

   Puis, en utilisant le fait que A soit une matrice symétrique,et définie positive, on a :

$= \frac{1}{2}\langle Ax_n, x_n\rangle + \alpha_n\langle Ax_n, d_n\rangle + \frac{\alpha_n^2}{2}\langle Ad_n, d_n\rangle - \langle b, x_n\rangle - \alpha_n\langle b, d_n\rangle$

On dérive par rapport à α_n :

$\frac{\partial J}{\partial \alpha_n}(x_{n+1}) = \langle Ax_n, d_n\rangle + \alpha_n\langle Ad_n, d_n\rangle - \langle b, d_n\rangle$

On peut donc dire que :

$\alpha_n = \frac{\langle b - Ax_n, d_n\rangle}{\langle Ad_n, d_n\rangle} \quad or \quad r_n = b - Ax_n$

D'où : $\alpha_n = \frac{\langle r_n, d_n\rangle}{\langle Ad_n, d_n\rangle}$
On remarque que : $r_n = b - Axn$
Donc :
$rn + 1 = b - Ax_{n+1}$
$= b - A(x_n + \alpha_n d_n)$
$= b - Ax_n - A\alpha_n d_n$
$= r_n - A\alpha_n d_n$

2. **Montrer que** $\beta_{n+1} = -\langle Ar_{n+1}, d_n\rangle / \langle d_n, Ad_n\rangle$
$d_{n+1} = r_{n+1} + \beta_{n+1} d_n$
$\Leftrightarrow \mathrm{d}_{n+1} - r_{n+1} = \beta_{n+1} d_n \Rightarrow \Phi(d_{n+1} - r_{n+1}) = \Phi(\beta_{n+1} d_n)$ Avec $\Phi(x) = \frac{\langle Ax, d_n\rangle}{\langle Ad_n, d_n\rangle}$
$\Rightarrow \langle Ad_{n+1}, d_i\rangle \frac{}{\langle Ad_n, d_n\rangle} - \frac{\langle Ar_{n+1}, d_i\rangle}{\langle Ad_n, d_n\rangle} = \frac{\langle \beta_{n+1} Ad_n, d_i\rangle}{\langle Ad_n, d_n\rangle}$
Or :

$\frac{\langle Ad_{n+1}, d_i\rangle}{\langle Ad_n, d_n\rangle} = 0$
et

$\frac{\langle \beta_{n+1} Ad_n, d_i\rangle}{\langle Ad_n, d_n\rangle} = \beta_{n+1}$

On a donc finalement :

$\beta_{n+1} = -\frac{\langle Ar_{n+1}, d_i\rangle}{\langle Ad_n, d_n\rangle}$

6 Algorithme du GC et complexité :

Pseudocode 1 Gradient Conjugué

Données : x_0, A, b
Initialisation : $r_0 = b - Ax_0$, $d_0 = r_0$
$i = 1$
tant que $(||r|| > \epsilon) ET (i > N)$ **faire**
  $z_i = Ad_i$
  $\alpha_i = \frac{\langle r_i, d_i\rangle}{\langle d_i, z_i\rangle}$
  $x_{i+1} = x_i + \alpha_i d_i$
  $r_{i+1} = r_i - \alpha_i z_i$
  $\beta_i = \frac{\langle r_i, z_i\rangle}{\langle d_i, z_i\rangle}$
  $d_{i+1} = r_{i+1} + \beta_i d_i$
  $k = k + 1$
fin tant que

Étudions maintenant la complexité algorithmique en temps et en espace. En terme d'espace sa complexité est limitée. Cet algorithme demande le stockage de 3 réel, ϵ, α et β ce qui fait donc $3 * 8$ octets. Il lui faut aussi stocké 4 vecteurs (x,b,r et d) de taille N soit $8 * 4 * N$ octets. Enfin il lui faut stocké une matrice A en format CSR. Chaque éléments de A non nul nécessite : 8 octets pour son coefficient, 4 octets pour l'indice de la colonne, 8 octets pour le pointeur vers l'élément suivant et enfin 4 octets pour la structure Ligne elle même. Soit au total 24 octets par élément. Notons $n_{A\neq 0}$ les coefficient non nul de A. On a donc besoin de 24*$n_{A\neq 0}$ octets pour stocké l'ensemble de notre matrice CSR.Nous avons donc au final une complexité algorithmique de $24 * n_{A\neq 0}+3 * 8+8 * 4 * N = 8(3 * n_{A\neq 0} + 4N + 3)$ octets

Sa complexité en terme de temps est aussi limitée. Comme A est symétrique définie positive et à valeur dans $\mathbb{R}$ (eq A est hermitienne) le produit scalaire $\langle Ar_i, d_i\rangle$ devient $\langle r_i, z_i\rangle$ avec $z_i = Ad_i$. Nous n'avons donc besoin que d'un seul produit matrice vecteur par iteration. Or on sait que cet algorithme converge en maximum N iteration. Comme A est stocké en CSR, ce produit represente la multiplication des $n_{A\neq 0}$ coefficient non nul par un coefficient du vecteur multiplié. Ainsi la complexité algoritmique du gradient conjugué est en $\mathcal{O}(n_{A\neq 0} * N)$.

7 Fonction GC :

Implémentons l'algorithme vu précedament en C. Nous aurons besoin pour cela de la bibliothèque MKL. Il se definie comme ci-dessous :

```
void GC(typeLigne *A,double *x,double *b, int n){

        //declaration variables locales
    float eps =1e-12;
    float alpha=0;
    float beta=0;
    double *Ax;
    double *tmp;
    double* r;
    double* d;
    double* z;
    double dAd;
    int k;
    int i;
    //init des variable locales
    dAd =0.;
    k=0.;
    tmp=(double*)malloc(n*sizeof(double));
    Ax=(double*)malloc(n*sizeof(double));
    r=(double*)malloc(n*sizeof(double));
    d=(double*)malloc(n*sizeof(double));
    z=(double*)malloc(n*sizeof(double));
    for (i=0;i<n;i++){
        r[i]=0.;
        d[i]=0.;
        z[i]=0.;
        tmp[i]=0.;
    }
    matveCSR(&A[0],&x[0],&Ax[0],n);
```

On initialise ensuite notre algorithme :

```
    //initialisation du GC
    cblas_dcopy(n,b,1,r,1);              //r=b
    cblas_daxpy(n,-1,Ax,1,r,1);          //r = r-A*x
    cblas_dcopy(n,r,1,d,1);              //d=r
```

Nous allons maintenant faire une boucle while qui se termine lorsque la norme de $r > \epsilon$ ou quand $k => n$) (rmq : ici on utilise => car la première itération est pour $k = 0$ et donc la n-ieme pour $k = n - 1$)

```
    while ((cblas_dnrm2(n,r,1) > eps)&&(k<n)){


        matveCSR(&A[0],&d[0],&z[0],n);  //z=A*d
        dAd=cblas_ddot(n,d,1,z,1);
        alpha=cblas_ddot(n,r,1,d,1)/dAd;//alpha=<r,d>/<z,z>
        cblas_daxpy(n,alpha,d,1,x,1);   //x=x+alpha*d
        cblas_daxpy(n,-alpha,z,1,r,1);  //r=r-alpha*z
        beta=-(cblas_ddot(n,r,1,z,1)/dAd);//beta=(r,z)/(d,z)
        cblas_dcopy(n,d,1,tmp,1);//on stock d dans un tampon et
        cblas_dcopy(n,r,1,d,1);  //on ecrase d=r afin de pourvoir faire :
```

```
        cblas_daxpy(n,beta,tmp,1,d,1);//d=r+beta*z  (restriction de
            nomenclature /mkl)
        k=k+1;
    }
}
```

Vérifions maintenant que notre algorithme à bien été implémenter. Prennons une matrice A inversible, un vecteur b très simple et calculons $x = A^{-1}b$. Soit

$$A = \begin{pmatrix} 4 & 0 & 2 \\ 0 & 5 & -3 \\ 2 & -3 & 6 \end{pmatrix} \tag{2}$$

$$b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{3}$$

d'où :

$$A^{-1} = \begin{pmatrix} \frac{21}{64} & \frac{-3}{32} & \frac{-5}{32} \\ \frac{-3}{32} & \frac{5}{16} & \frac{3}{16} \\ \frac{-5}{32} & \frac{3}{16} & \frac{5}{16} \end{pmatrix} \tag{4}$$

on obtient donc :

$$x = \begin{pmatrix} \frac{5}{64} \\ \frac{13}{32} \\ \frac{11}{32} \end{pmatrix} \approx \begin{pmatrix} 0.078 \\ 0.406 \\ 0.343 \end{pmatrix} \tag{5}$$

On utilisera le code suivant

```
int i;
int N = 3;
typeLigne *A;
double* x0;
double* b;
double* r;
double* d;
double* z;
//initialisation des variables
A=(typeLigne*)malloc(N*N*sizeof(typeLigne));
x0=(double*)malloc(N*sizeof(double));
b=(double*)malloc(N*sizeof(double));

for (i=0;i<N;i++)
{
    x0[i]=0.;
    b[i]=1.;
    A[i].A=0.;
    A[i].C=-1.;
    A[i].ptsuiv=NULL;
}

//on prend b=(1,1,1) (vect colone) pour avoir des calcul simple

//init de A
insert(&A[0],0,0,4.0);insert(&A[0],0,1,0.0);insert(&A[0],0,2,2.0) ;

insert(&A[0],1,0,0.0);insert(&A[0],1,1,5.0);insert(&A[0],1,2,-3.0);

insert(&A[0],2,0,2.0);insert(&A[0],2,1,-3.0);insert(&A[0],2,2,6.0)
   ;
    printf("Matrice␣A␣:␣\n");
    printMat(A,N);
```

```
    printf("\n");

    // on applique le GC
    GC(A,x0,b,N);

    for (i=0;i<N;i++)
        {
            printf("%lf\n",b[i]);
        }
    printf("\n");
    print
    for (i=0;i<N;i++)
    {
        printf("%lf\n",x0[i]);

    }

    //Liberation de la memoire allouee a A
    for(i = 0; i < N; i++)
        {
        freeLigne(&A[i]);
    }
```

Et on obtient la sortie suivante :

```
    Matrice A :
( 4.000000 0.000000 2.000000 )
( 0.000000 5.000000 -3.000000 )
( 2.000000 -3.000000 6.000000 )

1.000000
1.000000
1.000000

0.078125
0.406250
0.343750
```

Rmq : lorsque l'on change les conditions de la boucle while comme ceci :

```
    while ((cblas_dnrm2(n,r,1) > eps)&&(k<n+10))
```

les resultats restent identiques ce qui veut donc dire que notre algorithme converge bien en N étapes, N étant ici 3.

8 Solutions Manufacturées :

L'objectif de cette partie est de verifier si notre algorithme du GC fonctionne correctement pour des problèmes plus complexe. Nous allons le tester sur le problème du Laplacien 2D en difference finies d'ordre 2 sur un maillage régulier aux conditions limites homogènes. Ceci utilisé pour approché une solution de l'équation différentiel de type $u'' = f$.

Pour implémenter cela nous allons avoir besoin tout d'abord de choisir une fonction u afin de pouvoir calculer $\frac{\partial u^2}{\partial^2 x} + \frac{\partial u^2}{\partial^2 x} = f$. Pour choisir ce u il faut aussi prendre en compte les conditions aux limites de Diriclet : nous choisissons la fonction $u(x, y) = \sin(2\pi x)\sin(2\pi y)$. On a donc $f(x, y) = 8\pi^2 u(x, y)$

```
double u(double x, double y){
    return sin(2.*M_PI*x)*sin(2.*M_PI*y);}

double f(double x, double y){
    return -8.*(M_PI*M_PI)*u(x,y);
}
```

Rmq : MPI est une macro definie en debut de programme comme ceci

```
#define M_PI 3.14159265358979323846
```

Rmq_2 : cette fonction u est intéressante car facilement derivable deux fois selon x et y ainsi que constante (null dans ce cas) sur les bords, on respect donc bien les conditions de Diriclet.

Cependant nous allons devoir discretiser ("mettre sous forme de vecteur") c'est deux fonction afin de pouvoir utiliser la fonction GC. Pour ce faire on utilise la fonction suivante cette fonction prendra en paramêtre un pointeur sur fonction prenant en entré deux doubles et renvoyant un double afin de pouvoir utiliser la fonction "discretisation" pour u et pour f :

```
double* Discretisation(double (*f)(double,double),int Nx, int Ny){
    //variable locales
    int i;
        int j;
        int I;
    double hx;
        double hy;
    double *fdiscretise;
    //creation de notre vecteur pour stocker f sous forme discrete
    fdiscretise=(double*)malloc(Nx*Ny*sizeof(double));
    for (i=0;i<Nx*Ny;i++){fdiscretise[i]=0.;}
    //init des parametre de discretisation
    hx=1./((double)Nx+1.);
    hy=1./((double)Ny+1.);
    //discretisation
    for (i=0;i<Nx;i++)
    {
        for (j=0;j<Ny;j++)
        {
        I=i+j*Nx;
        fdiscretise[I]=f((double)(i+1)*hx,(double)(j+1)*hy);
        }
    }
    //revoie de notre fonction discretisee
    return fdiscretise;
}
```

Nous aurons aussi besoin de crée une fonction pour générer le Laplacien 2D. Elle s'écrit sous cette forme :

```
void Laplacien2D(typeLigne *MatA,int Nx, int Ny){
        //fonction de construction du laplacien 2D
    int N=Nx*Ny;
    int i,j,I;
    double hx,hy;
    double inv_hx_2,inv_hy_2;
    //creation du maillage
    hx=1./(double)(Nx-1);
    hy=1./(double)(Ny-1);
    inv_hx_2=1./(hx*hx);
    inv_hy_2=1./(hy*hy);
    //remplissage de A
    for (j=0;j<Ny;j++){
            for (i=0;i<Nx;i++){
                I=i+j*Nx;
                insert(&MatA[0],I,I,-2*(inv_hx_2+inv_hy_2));//A_{I,I}

                if(I>0){
                        insert(&MatA[0],I,I-1,inv_hx_2);//A_{I,I-1}
                }
                if(I<N){
                        insert(&MatA[0],I,I+1,inv_hx_2);//A_{I,I+1}
                }
                if(I>=Nx){
                        insert(&MatA[0],I,I-Nx,inv_hy_2);//A_{I,I-Nx}
                }
                if(I<=N-Nx){
```

```
                        insert(&MatA[0],I,I+Nx,inv_hy_2);//A_{I,I+Nx}
                }
            }
    }
}
```

Enfin nous aurons besoin de d'une fonction pour calculer la norme infinie de $u_{exacte} - u_{approx}$ pour verifier la validité de notre implementation.

```
double normeinf(double *x,int N){
    int i;
    double max;
    max=ABS(x[0]);
    for(i=0;i<N;i++){
        if (ABS(x[i])>max){max=ABS(x[i]);}
    }
    return max;
}
```

rmq : dans cette fonction nous defons utiliser la valeur absolue. Dans <math.h>, cette fonction est definie uniquement pour des entiers, nous devons donc crée la notre comme ceci :

```
double ABS(double a){
        if (a<0) return -a;
        else return a;
}
```

Maintenant que nous avons toutes nos fonctions, nous pouvons commencer à implementer la méthode des solutions manufacturées qui prendra en entré deux entier N_x et N_y :

```
void SolutionManufactur e(int Nx, int Ny)
{
    //def et init des variables locales
    int i;
    typeLigne *A;
    double *uapprox;
    double *udis;
    double *fdis;
    int N = Nx*Ny;
    uapprox=(double*)malloc(N*sizeof(double));
    A=(typeLigne*)malloc(N*N*sizeof(typeLigne));
    for (i=0;i<N;i++)
    {
        initLigne(&A[i]);
    }
    //discretisation de u et f
    udis=Discretisation(u,Nx,Ny);
    fdis=Discretisation(f,Nx,Ny);
    //g n ration du Laplacien 2D
    Laplacien2D(A,Nx,Ny);
    //application du GC sur A et fdis pour trouver uapprox
    GC(A,uapprox,fdis,Nx*Ny);
    //norme infinie de uapprox-udis
    for (i=0;i<N;i++)
    {
        uapprox[i]=uapprox[i]-udis[i];
    }
    printf("Norme␣infinie␣:␣%lf␣",normeinf(uapprox,N));
}
```

Nous obtenons en sortie pour $N_x = N_y = 100$:

```
Norme infinie : 0.054689
```

Nous obtenons en sortie pour $N_x = N_y = 10$:

```
Norme infinie : 0.667423
```

On voit donc bien que plus on est precis dans notre discretisation (i.e. N_x, $N_y \longrightarrow +\infty$) plus notre u_{approx} est proche de u_{exact} (i.e. $|u_{exacte} - u_{approx}|_\infty \longrightarrow 0$). Notre implementation est donc valide.