Rapport TP1 - Algorithmique Numerique
Partie 2 : Méthodes Iteratives

YMERAJ Eldis

10/2022

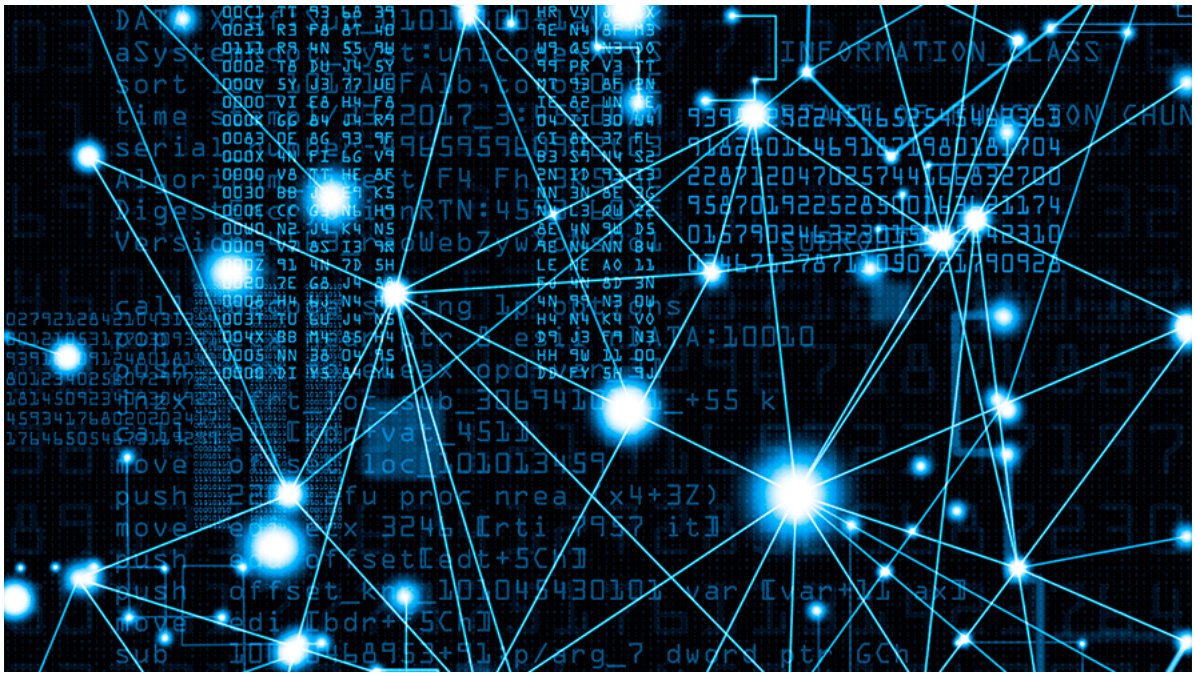

Figure 1 – Algorithmique numerique

Table des matières

Introduction aux Méthodes iteratives

Nous allons étudier les méthodes itératives pour résoudre un système d'équations linéaires Ax = b.

Une méthode itérative engendre une suite de vecteurs qui doit tendre vers la solution de l'équation Ax = b.

Dans les méthodes de Jacobi et de Gauss-Seidel, le passage d'un vecteur x(k) de la suite au suivant se fait en corrigeant successivement (ou simultanément) une (ou plusieurs) composante de ce vecteur. Ces corrections s'effectuent en annulant une (ou plusieurs) composante du résidu r(k) = b - Ax(k)
Dans cette partie de TP je vais implémenter les méthodes de Jacobi et Gauss-Seidel pour la résolution de systèmes linéaires dans le cas des matrices tests proposées dans le section 4 de TP.

1 Rappel - Méthode de Jacobi

La méthode de Jacobi, due au mathématicien allemand Karl Jacobi, est une méthode itérative de résolution d'un système matriciel de la forme Ax = b. Pour cela, on utilise une suite x(k) qui converge vers un point fixe x, solution du système d'équations linéaires.

1.1 Algorithme de Jacobi

- L'algorithme de Jacobi s'écrit pour une précision epsilon
- Choisir un point initial x(0).
- Calculer l'erreur epsilon(0) = norme(A*x(0) - b).
- poser k = 0.

Tant que $(\epsilon^{(k)} \geq \epsilon)$
$x_i^{(k+1)} = \frac{1}{a_{ii}}[b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}], i = 1, \ldots, n$
$\epsilon^{(k+1)} = ||Ax^{(k+1)} - b||$;
k=k+1 ;
Fin Jacobi

Figure 2 – Algorithme de Jacobi

2 Rappel - Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est une méthode itérative de résolution d'un système linéaire (de dimension finie) de la forme Ax=b, ce qui signifie qu'elle génère une suite qui converge vers une solution de cette équation, lorsque celle-ci en a une et lorsque des conditions de convergence sont satisfaites (par exemple lorsque A est symétrique définie positive). L'algorithme suppose que la diagonale de A est formée d'éléments non nuls.

2.1 Algorithme de Gauss-Seidel

- L'algorithme de Jacobi s'écrit pour une précision epsilon
  - Choisir un point initial x(0).

— Calculer l'erreur epsilon(0) = norme(A*x(0) - b).
— poser k = 0.

Tant que $(\epsilon^{(k)} \geq \epsilon)$
$x_i^{(k+1)} = \frac{1}{a_{ii}}[b_i - \sum_{j_{(i>j)}} a_{ij}x_j^{(k+1)} - \sum_{j_{(i<j)}} a_{ij}x_j^{(k)}], i = 1, \ldots, n$
$\epsilon^{(k+1)} = ||Ax^{(k+1)} - b||$;
k=k+1 ;
Fin Gauss-Seidel

Figure 3 – Algorithme de Gauss-Seidel

3 Implementation en langage C

Dans cette partie je vais vous montrer comment j'ai implementé la méthode de Jacobi et Gauss-Seidel en langage C et toutes les fonctions que j'ai construit pour réaliser ce travail.

3.1 Fonction allouerMat(n, m)

Cette fonction permet de nous allouer la memoire pour les matrices qu'on va utiliser.

```c
float ** allouerMat(int n, int m)
{
    int i;
    float ** mat = NULL;
    mat = malloc(n*sizeof(float *));
    if(mat == NULL){return NULL;}

    for(i=0;i<n;i++)
    {
        mat[i] = NULL;
        mat[i] = malloc(n *sizeof(float));
        if (mat[i] == NULL)
            {
                printf("erreur d'allocation");

             return NULL;}
    }
    return mat;
}
```

— La fonction prends une valeur en retour et il va nous retourner une matrice alloué.
— Prends comme argument les dimensions de la matrice qu'on va allouer.
— Si notre matrice est vide alors la fonction va nous retourner rien.

3.2 Fonction entrerMatrice(A, n, m)

Cette fonction permet à l'utilisateur d'entrer la valeur de matrice qu'il va tester. Dans notre cas sont les matrices A1, A2, A3, A4. Après qu'on a alloué l'espace pur la matrice qu'on va tester on demande à

l'utilisateur de remplir la matrice.

```c
void entrerMatrice(float **A, int n, int m)
{
   printf("Entrez les valeurs de la matrice quoe vous voulez tester\n");
    for(int i =0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            scanf("%f", &A[i][j]);
        }
    }
}
```

— Cette fonction est de type void, qui ne retourne rien.
— Elle prend comme argument une matrice de type float qu'on a nommée A, et ses dimensions.
— Demande à l'utilisateur d'entrer les valeurs de matrice.

3.3 Fonction affMat(A, n, m)

Cette fonction permet d'afficher une matrice.

```c
void affMat(float **A, int n, int m){

    int i, j;
    for(i=0; i<n; i++){
        for(j=0; j<m; j++){
            printf("%.2f \t", A[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

— Cette fonction est de type void, qui ne retourne rien.
— Elle prend comme argument une matrice de type float qu'on a nommée A, et ses dimensions.
— Elle va nous afficher la matrice avec ses valeurs.

3.4 Fonction affVect(A, n, m)

Cette fonction permet de realiser l'affichage d'un vecteur de dimension n.

```
void affVect(float *A, int n){

    int i, j;
    for(i=0; i<n; i++){
            printf("%.2f \t", A[i]);

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

- Cette fonction est de type void, qui ne retourne rien.
- Elle prend comme argument un vecteur de type float et son dimension.
- Elle va nous afficher le vecteur qu'on veut avec ses valeurs.

3.5 Fonction trouveBjac(mat, int n)

Cette fonction permet de nous trouver la matrice B qui est de dimension 3x1. Le vecteur B est choisie de sorte que la solution exacte du système d'équations est x[i]=1, pour i =1,...,n.

```
float * trouveBjac(float **mat, int n)
{

    float *B=malloc(n*sizeof(float));

    for(int i = 0; i < n; i++)
    {
        B[i] = 0;

        for(int j = 0; j < n; j++)
        {

            B[i]+= mat[i][j];
        }
    }

    return B;
}
```

- Cette fonction renvoit un vecteur de type float.
- Elle prend comme argument la matrice test et sa diemnsion.
- Elle permet de trouver le vecteur B telle que lq solution exacte est x[i] = 1.

3.6 remplMatA5(A5, n)

Cette fonction va nous servir pour les matrices test, elle permet de remplir la matrice A5 en suivant la régle donnée.

```
void remplMatA5(float ** A5, int n){
    int i, j;
    for(i = 0; i < n; i++)
    {
        A5[i][i] = 1;
    }
    for(j = 1; j < n; j++)
    {
        A5[1][j] = A5[j][1] = 1/((float)pow(2,(float)j)); //ici le pow permet de mettre 2 à
    la puissance j
    }
}
```

— Cette fonction est de type void, qui ne retourne rien.
— Elle prend comme argument la matrice test A5 qu'on a déclaré et alloué dans le main et le dimension n (on va utiliser que la dimension n car on utilise des matrices carrée nxn).
— Elle permet de remplir la matrice A5 en suivant la régle donnée dans la fiche de TP.

3.7 remplMatA6(A6, n)

Cette fonction va nous servir pour les matrices test, elle permet de remplir la matrice A6 en suivant la régle donnée.

```
void remplMatA6(float **A6, int n)
{
    for(int i =0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            if(i==j)
            {
                A6[i][j]=3;
            }
            else if(j==i+1 && i<n)
            {
                A6[i][j]=-1;
            }
            else if(j==i-1 && i>1)
            {
                A6[i][j]=-2;
            }else
            {
                A6[i][j]=0;
            }
        }
    }

}
```

— Cette fonction est de type void, qui ne retourne rien.
— Elle prend comme argument la matrice test A6 qu'on a déclaré et alloué dans le main et le dimension

n (on va utiliser que la dimension n car on utilise des matrices carrée nxn).
— Elle permet de remplir la matrice A5 en suivant la régle donnée dans la fiche de TP.

3.8 Fonction Norme(vect, int n)

Cette fonction permet de calculer la norme d'un vecteur.

```
1  float Norme(float *vect, int n)
2  {
3
4      float N = 0.0;
5      for(int i =0; i<n; i++)
6      {
7          N+= vect[i]*vect[i] ;
8      }
9
10     return sqrt(N);
11 }
```

— Cette fonction est de type float et elle va nous retoutner une valeur flottant.
— Elle prend comme argument un vecteur de type float et son dimension.
— Premierement elle calcule la somme de tout les élements au carrée du vecteur.
— A la fin, elle nous retourne la racine carré de cette somme, qui est la norme.

3.9 Fonction initX(n)

Cette fonction permet d'initialiser un vecteur à 0 .

```
1  float *initX(int n)
2  {
3      float *x = malloc(n*sizeof(float));
4      for(int i =0; i<n; i++)
5      {
6          x[i] = 0.0;
7      }
8      return x;
9  }
```

— Cette fonction renvoit un vecteur de type float.
— Elle prend comme argument la dimension de vecteur qu'on va initialiser à 0.
— Elle nous retourne le vecteur initialisé à 0.

3.10 Fonction multipl(A, b, n)

Fonction qui permet de faire la multiplication entre une matrice carrée et un vecteur.

```
float *multipl(float **A, float *b, int n)
{
    float *mult = malloc(n*sizeof(float));
    for(int i=0; i<n;i++)
    {
        mult[i] = 0.0;
        for(int j=0; j<n; j++)
        {
            mult[i]+= A[i][j]*b[j];
        }
    }
    return mult;
}
```

— Cette fonction renvoit un vecteur de type float.
— Elle prend comme argument la matrice carree et vecteur qu'on va multiplier et leur dimension.
— La fonction va nous retourner le vecteur qui tient la valeur de multiplication.

3.11 Fonction soustract(v1, v2, n)

Fonction qui permet de faire la soustraction entre deux vecteurs.

```
float *soustract(float *v1, float *v2, int n)
{
    float *soustrac = malloc(n*sizeof(float));
    for(int i =0; i<n; i++)
    {
        soustrac[i] = 0.0;
        soustrac[i]+= v1[i] - v2[i];
    }
    return soustrac;

}
```

— Cette fonction renvoit un vecteur de type float.
— Elle prend comme argument deux vecteurs v1, v2 et leur dimension<
— La fonction fait la soustraction de vecteurs v1 et v2 de maniere suivante v1-v2.
— La fonction va nous retourner un vectur = v1-v2.

3.12 Fonction calcEpsi(A, b, x0, n)

Fonction qui permet de nous calculer la valeur d'epsilon pour l'algorithme de Jacobi et Gauss-Seidel.

```
float calcEpsi(float **A, float *b, float*x0, int n)
{
    float *epsi;
    epsi = malloc(n*sizeof(float));
    for(int i =0;i<n;i++)
    {
        epsi[i] = 0.0;
    }

    epsi = multipl(A, x0, n);

    epsi = soustract(epsi, b, n);

    return Norme(epsi, n);
}
```

— Elle prend comme argument une matrice, le vecteur b, le vecteur x0 et la dimension n.
— La fonction calcule la valeur d'epsilon grace à fonctions multipl(A, b ; n), soustract(v1, v2, n) et la fonction Norme(vect, n).
— Cette fonction renvoit une valeur folttant qui est la valeur de notre epsilon.
— Les calcules sont faites en suivant l'algorithme de Jacobi et Gauss-Seidel.

3.13 Fonction xk(A, b, x0, n, i)

Fonction qui permet de calculer la valeur de x[i] à l'iteration (k+1), qui est la premiere partie de l'algorithme de Jacobi.

```
float xk(float **A, float *b, float *x0, int n, int i)
{
    float a;

    float XK ;

    float sum = 0;
     a = 1/A[i][i];

    for(int j =0; j<n; j++)
    {
        if(i != j)
          {
             sum = sum + A[i][j]*x0[j];
          }
    }
    sum = b[i] - sum;
    XK = a*sum;

    return XK;

}
```

— Elle prend comme argument une matrice test , le vecteur b, vecteur x0, leur dimension n et un entier i

— La fonction calcule la valeur d'epsilon grace à fonctions multipl(A, b ; n), soustract(v1, v2, n) et la fonction Norme(vect, n).
— Cette fonction renvoit une valeur folttant qui est notre x[i](k+1).

3.14 Fonction errRel(x0)

Fonction qui permet de calculer l'erreur relative entre deux solutions en suivant la formule donné à la fiche de tp.

```
float errRel(float *x0)
{
    float err1,err2,err3;
    err1=1-x0[0];
    err2=1-x0[1];
    err3=1-x0[2];

    if ((err1>=err2)&&(err1>=err3))
    {
        return (err1);
    }
    else if((err2>=err1)&&(err2>=err3))
        {
            return(err2);
        }
        else
        {
            return (err3);
        }


}
```

— Elle prend comme argument un vecteur qui est solution du systeme.
— Cette fonction renvoit la valeur calculé de l'erreur relative en suivant la formule.

3.15 Fonction Jacobi(A, b, x0, n, epsi)

Fonction qui realise la methode de Jacobi en suivant l'algorithme donnee en cours. On est arrivé à écrire le code de Jacobi à l'aide de fonctions précédents qu'on a crée et utilisé dans cette fonction.

```
int Jacobi(float **A, float *b, float *x0, int n, float epsi)
{
    int   k = 0;
    float *x0k1 =malloc(n*sizeof(float));
    float epsi1 = 1;

    while( (epsi1 >= epsi)  && (k <50) )
    {
        for(int i=0;i<n;i++)
        {
            x0k1 [i]= xk(A, b, x0, n, i);
        }


        epsi1 = calcEpsi(A, b, x0k1, n);
        x0 = x0k1;
        printf("L'erreur relative de l'iteration %d est : %f \n", k+1, errRel(x0));
```

```
18          k = k+1;
19
20      }
21      return k;
22  }
```

— Elle prend comme argument La matrice test, le vecteur b, le vecteur x0, leur dimension et epsilon.
— La fonction applique étape par étape l'algorithme de Jacobi.
— Elle affiche l'erreur relative après chaque itération.
— Elle nous retourne le nombre d'itérations pour cette méthode.

3.16 Fonction GS(A, b, x0, n, epsi)

Fonction qui realise la methode de Gauss-Seidel en suivant l'algorithme donnee en cours. On est arrivé à écrire le code de Jacobi à l'aide de fonctions précédents qu'on a crée et utilisé dans cette fonction.

```
1   int GS(float **A, float *b, float *x0, int n, float epsi)
2   {
3       int   k = 0;
4       float epsi1 = 1;
5
6       while( (epsi1 >= epsi) && (k <50) )
7       {
8           for(int i=0;i<n;i++)
9           {
10              x0[i] = xk(A, b, x0, n,i);
11          }
12
13          epsi1 = calcEpsi(A, b, x0, n);
14          printf("L'erreur relative de l'iteration %d est : %f \n", k+1, errRel(x0));
15
16          k = k+1;
17
18      }
19
20      return k;
21  }
```

— Elle prend comme argument La matrice test, le vecteur b, le vecteur x0, leur dimension et epsilon.
— La fonction applique étape par étape l'algorithme de Gauss-Seidel.
— Elle affiche l'erreur relative après chaque itération.
— Elle nous retourne le nombre d'itérations pour cette méthode.

4 La fonction main()

Dans le main on a testé les matrices test A1, A2, A3, A4, A5, A6 et on a obtenu les résultats qu'on va expliquer après.

Les étapes qu'on a suivi sont :
— Premièrement on a déclaré les dimentions nxm et tant que on a que des matrices carrée de dimensions 3x3 on a donnee la meme valeur pour n et m =3.
— On a déclaré une variable epsi de typle float, qui est égale à 0.001.

— On a déclaré la matrice qu'on va tester et on l'a alloué la memoire à l'aide de fonction allouerMat(n, m).
— Après à l'aide de la fonction entrerMatrice(A, n, m) on demande à l'utilisateur de remplir la matrice avec les valeurs de matrice test donnée à la fiche de TP.
— On déclare une variable x0 et on l'a initialisé à 0 à l'aide de fonction initX(n).
— Maintement on a déclaré le vecteur B et on a alloué la memoire pour ce vecteur.
— A l'aide de fonction trouveBjac on a initialisé le vecteur B et on l'affiche.
— Après on a déclaré une variable de type int qu'on a nommée nbIter et on a fait l'appel à fonction Jacobi(A, B, x0, n, epsi) pour stocker le nombre d'itérations par la méthode de Jacobi dans cette variable.
— On a fait la meme chose pour Gauss-Seidel en declarant une variable nbGS et e faissant appel à fonction GS(A, B, x0, n, epsi).
— Et à la fin on a fait l'affichage de nombres d'itérations pour chque méthode.
— Pour les matrices A5 et A6 on a mis en commentaire les demarches.
— C'est la meme chose mais on remplit les matrices avec les fonctions remplMatA5(A5, n) et remplMatA6(A6, n).
C'est la même procédure pour tout les matrices test. Et pour tester les matrices l'utilisateur entre les valeurs de la matrice test qui va tester et il va obtenir le résultat.
Et si on veut de testerles matrices A5 et A6 "separement" on juste enlève les "/* */" et o met en commentaire le code qui n'est pas en commentaire actuellement.

```c
int main()
{
    int     n = 3;
    int     m = 3;
    float   epsi = 0.001;

    printf("\n==========MATRICE TEST==========\n");

    float **A = allouerMat(n,n);
    entrerMatrice(A, n, m);
    printf("La matrice que vous avez entree est: \n");
    affMat(A, n, m);

    float *x0 = malloc(n*sizeof(float));
    // initialisation de vecteur x0 = 0
    x0 = initX(n);

    float *B = malloc(n*sizeof(float));
    B = trouveBjac(A, n);
    printf("Le vecteur B pour que xi est toujours egal a 1 est : \n");
    affVect(B, n);

    int nbIter = Jacobi(A, B, x0, n, epsi);
    printf("\n==>Nombre d'iterations pour jacobi est: %d\n\n", nbIter);

    int nbGS = GS(A, B, x0, n, epsi);
    printf("\n==>Nombre d iterations pour Gauss-Seidel est: %d\n", nbGS);


    /*
    printf("==========Matrice test A5==========");
    float **A5 = allouerMat(n,n);
    remplMatA5(A5,n);
    printf("\nNotre matrice A5  est: \n");
    affMat(A5, n, m);

    float *x0 = malloc(n*sizeof(float));
    // initialisation de vecteur x0 = 0
    x0 = initX(n);
```

```
40
41      float *B5 = malloc(n*sizeof(float));
42      B5 = trouveBjac(A5, n);
43
44      printf("La matrice B5 pour que xi est toujours egal a 1 est : \n");
45      affVect(B5, n);
46
47      int nbIter = Jacobi(A5, B5, x0, n, epsi);
48      printf("\n==>Nombre d'iterations pour jacobi est: %d\n\n", nbIter);
49
50      int nbGS = GS(A5, B5, x0, n, epsi);
51      printf("\n==>Nombre d iterations pour Gauss-Seidel est: %d\n", nbGS);
52      */
53
54      /*
55      printf("==================Matrice test A6==================");
56      float **A6 = allouerMat(n,n);
57      remplMatA6(A6,n);
58      printf("\nNotre matrice A6  est: \n");
59      affMat(A6, n, m);
60
61      float *x0 = malloc(n*sizeof(float));
62      // initialisation de vecteur x0 = 0
63      x0 = initX(n);
64
65      float *B6 = malloc(n*sizeof(float));
66      B6 = trouveBjac(A6, n);
67
68      printf("La matrice B6 pour que xi est toujours egal a 1 est : \n");
69      affVect(B6, n);
70
71      int nbIter = Jacobi(A6, B6, x0, n, epsi);
72      printf("\n==>Nombre d'iterations pour jacobi est: %d\n\n", nbIter);
73
74      int nbGS = GS(A6, B6, x0, n, epsi);
75      printf("\n==>Nombre d iterations pour Gauss-Seidel est: %d\n", nbGS);
76      */
77
78      return 0;
79
80  }
```

5 Jeux d'essais

5.1 Matrice test A1

Pour la matrice test A1 on a obtenu le resultat suivant :

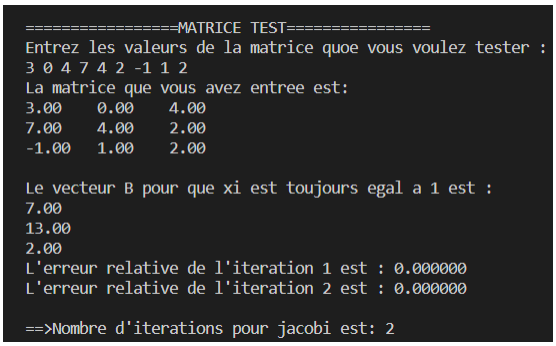

Figure 4 – Matrice A1 Jacobi

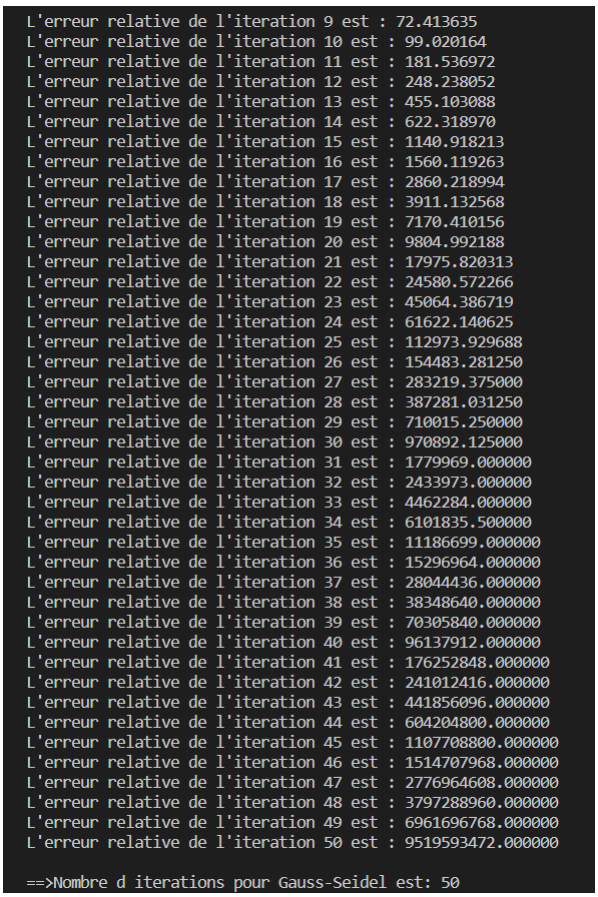

Figure 5 – A1 Gauss-Seidel

Pour la matrice A1 la methode de Gauss-Seidel ne converge pas, on a essayé pour 50 ou plus itérations.

5.2 Matrice test A2

Pour la matrice test A3 on a obtenu le resultat suivant :

```
L'erreur relative de l'iteration 29 est : 638167936.000000
L'erreur relative de l'iteration 30 est : -709075456.000000
L'erreur relative de l'iteration 31 est : 2552671744.000000
L'erreur relative de l'iteration 32 est : -2836301824.000000
L'erreur relative de l'iteration 33 est : 10210686976.000000
L'erreur relative de l'iteration 34 est : -11345207296.000000
L'erreur relative de l'iteration 35 est : 40842747904.000000
L'erreur relative de l'iteration 36 est : -45380829184.000000
L'erreur relative de l'iteration 37 est : 163370991616.000000
L'erreur relative de l'iteration 38 est : -181523316736.000000
L'erreur relative de l'iteration 39 est : 653483966464.000000
L'erreur relative de l'iteration 40 est : -726093266944.000000
L'erreur relative de l'iteration 41 est : 2613935865856.000000
L'erreur relative de l'iteration 42 est : -2904373067776.000000
L'erreur relative de l'iteration 43 est : 10455743463424.000000
L'erreur relative de l'iteration 44 est : -11617492271104.000000
L'erreur relative de l'iteration 45 est : 41822973853696.000000
L'erreur relative de l'iteration 46 est : -46469969084416.000000
L'erreur relative de l'iteration 47 est : 167291895414784.000000
L'erreur relative de l'iteration 48 est : -185879876337664.000000
L'erreur relative de l'iteration 49 est : 669167581659136.000000
L'erreur relative de l'iteration 50 est : -743519505350656.000000

==>Nombre d'iterations pour jacobi est: 50
```

Figure 6 – Matrice A2 Jacobi

```
L'erreur relative de l'iteration 10 est : 956.377808
L'erreur relative de l'iteration 11 est : -1062.642090
L'erreur relative de l'iteration 12 est : 3825.511230
L'erreur relative de l'iteration 13 est : -4250.568359
L'erreur relative de l'iteration 14 est : 15302.044922
L'erreur relative de l'iteration 15 est : -17002.273438
L'erreur relative de l'iteration 16 est : 61208.179688
L'erreur relative de l'iteration 17 est : -68009.093750
L'erreur relative de l'iteration 18 est : 244832.718750
L'erreur relative de l'iteration 19 est : -272036.375000
L'erreur relative de l'iteration 20 est : 979330.875000
L'erreur relative de l'iteration 21 est : -1088145.500000
L'erreur relative de l'iteration 22 est : 3917323.750000
L'erreur relative de l'iteration 23 est : -4352582.500000
L'erreur relative de l'iteration 24 est : 15669299.000000
L'erreur relative de l'iteration 25 est : -17410332.000000
L'erreur relative de l'iteration 26 est : 62677204.000000
L'erreur relative de l'iteration 27 est : -69641336.000000
L'erreur relative de l'iteration 28 est : 250708816.000000
L'erreur relative de l'iteration 29 est : -278565344.000000
L'erreur relative de l'iteration 30 est : 1002835264.000000
L'erreur relative de l'iteration 31 est : -1114261376.000000
L'erreur relative de l'iteration 32 est : 4011341056.000000
L'erreur relative de l'iteration 33 est : -4457045504.000000
L'erreur relative de l'iteration 34 est : 16045364224.000000
L'erreur relative de l'iteration 35 est : -17828182016.000000
L'erreur relative de l'iteration 36 est : 64181456896.000000
L'erreur relative de l'iteration 37 est : -71312728064.000000
L'erreur relative de l'iteration 38 est : 256725827584.000000
L'erreur relative de l'iteration 39 est : -285250912256.000000
L'erreur relative de l'iteration 40 est : 1026903310336.000000
L'erreur relative de l'iteration 41 est : -1141003649024.000000
L'erreur relative de l'iteration 42 est : 4107613241344.000000
L'erreur relative de l'iteration 43 est : -4564014596096.000000
L'erreur relative de l'iteration 44 est : 16430452965376.000000
L'erreur relative de l'iteration 45 est : -18256058384384.000000
L'erreur relative de l'iteration 46 est : 6572181186150 4.000000
L'erreur relative de l'iteration 47 est : -73024233537536.000000
L'erreur relative de l'iteration 48 est : 262887247446016.000000
L'erreur relative de l'iteration 49 est : -292096934150144.000000
L'erreur relative de l'iteration 50 est : 1051548989784064.000000

==>Nombre d iterations pour Gauss-Seidel est: 50
```

Figure 7 – Matrice A2 Gauss-Seidel

Pour la matrice test A2 la méthode de Jacobi et Gauss Seidel ne convergent pas, on a essayé pour 50 ou plus itérations.

5.3 Matrice test A3

Pour la matrice test A3 on a obtenu le resultat suivant :

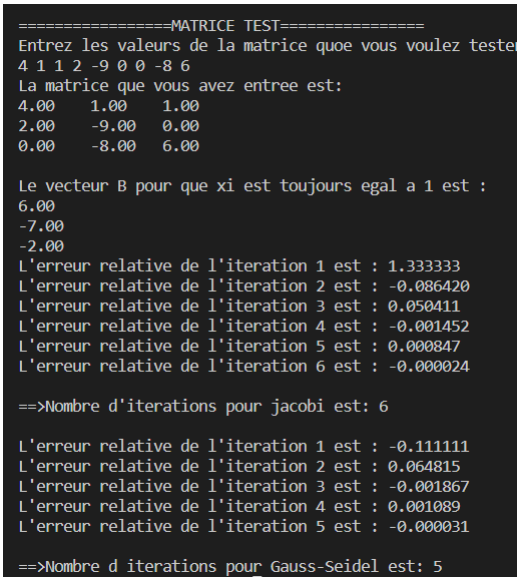

Figure 8 – Matrice A3 Jacobi ET Gauss-Seidel

Pour la matrice test A3, les deux méthodes convergent mais la méthode de Gauss-Seidel est plus rapide. L'erreur relative à l'itération 6 pour Jacobi est -0.000024 et pour Gaus-Seidel à l'iteration 5 est -0.000031.

5.4 Matrice test A4

Pour la matrice test A4 on a obtenu le resultat suivant :

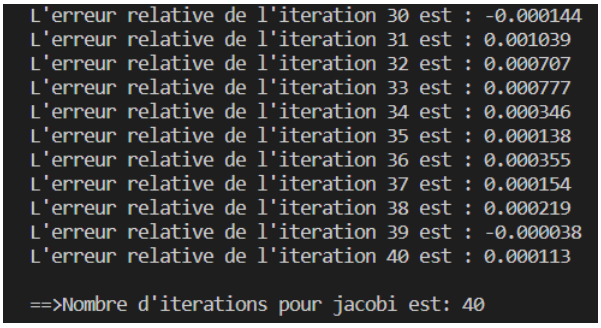

Figure 9 – Matrice A4 Jacobi

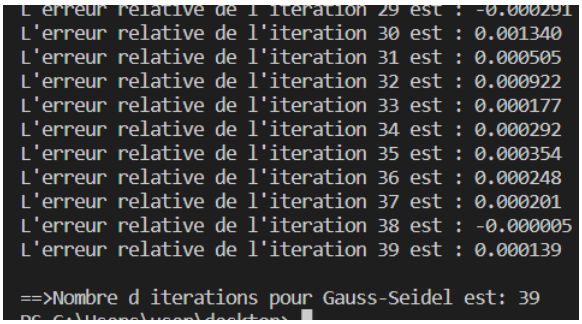

Figure 10 – Matrice A4 Gauss-Seidel

Pour la matrice test A4, les deux méthodes convergent mais la méthode de Gauss-Seidel est plus rapide. L'erreur relative à l'itération 40 pour Jacobi est 0.000113 et pour Gaus-Seidel à l'iteration 39 est 0.000139. Donc nombre d'iterations pour Jacobi est : 40 Pour Gauss-Seidel : 39

5.5 Matrice test A5

Pour la matrice test A5 on a obtenu le resultat suivant : Pour la matrice test A5, les deux méthodes

```
================Matrice test A5==============
Notre matrice A5  est:
1.00    0.00    0.00
0.00    0.50    0.25
0.00    0.25    1.00

La matrice B5 pour que xi est toujours egal a 1 est :
1.00
0.75
1.25
L'erreur relative de l'iteration 1 est : 0.000000
L'erreur relative de l'iteration 2 est : 0.125000
L'erreur relative de l'iteration 3 est : 0.015625
L'erreur relative de l'iteration 4 est : 0.001953

==>Nombre d'iterations pour jacobi est: 4

L'erreur relative de l'iteration 1 est : 0.125000
L'erreur relative de l'iteration 2 est : 0.015625
L'erreur relative de l'iteration 3 est : 0.001953
L'erreur relative de l'iteration 4 est : 0.000244

==>Nombre d iterations pour Gauss-Seidel est: 4
```

Figure 11 – Matrice A5 Jacobi et Gauss-Seidel

convergent, ils ont le meme nombre d'iterations. L'erreur relative à l'itération 4 pour Jacobi est 0.001953 et pour Gaus-Seidel à l'iteration 4 est 0.000244. Donc nombre d'iterations pour Jacobi est : 4 Pour Gauss-Seidel : 4

5.6 Matrice test A6

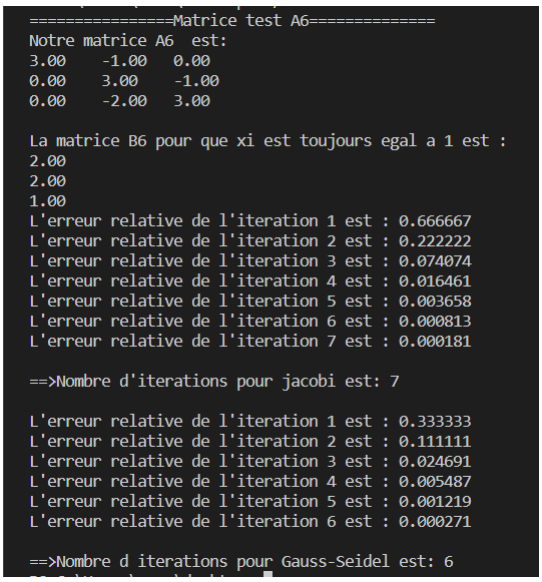

Figure 12 – Matrice A6 Jacobi et Gauss-Seidel

Pour la matrice test A6, les deux méthodes convergent mais la méthode de Gauss-Seidel est plus rapide. L'erreur relative à l'itération 7 pour Jacobi est 0.000181 et pour Gaus-Seidel à l'iteration 6 est 0.000271. Donc nombre d'iterations pour Jacobi est : 7
Pour Gauss-Seidel : 6

Conclusion

L'essentiel des méthodes itératives pour résoudre les systèmes linéaires sont issues d'une méthode de point fixe et leur convergence est assurée des que la matrice d'itération a un rayon spectral inférieur à l'unité. Elles présentent une proprieté importante : chaque itération permet se rapprocher de la solution. Ceci signifie que si on part proche de la solution, la convergence sera obtenue tres rapidement.

La méthode de Jacobi : Cette méthode n'est pas toujours bien définie. En effet, il suffit qu'au moins un élément diagonal soit nul pour que l'algorithme ne soit plus valide. Néanmoins, lorsque la matrice A est une matrice définie positive, ses coefficients diagonaux sont strictement positifs et nous pouvons donc appliquer l'algorithme Jacobi. Aussi lorsque A est à diagonale strictement dominante, la méthode est correctement définie et nous sommes même en mesure de présenter un résultat de convergence.

La méthode de Gauss-Seidel : On arrive en conclusion que la méthode de Gauss-Seidel converge dans le cas de matrices symétriques défiies positives. Et en géneral cette méthode est plus rapide que la méthode de Jacobi.

Comparaison des méthodes

- Si A est définie positive alors la méthode de Jacobi est définie. Si de plus, A est à diagonale strictement dominante alors la méthode de Jacobi converge
- Si A est symétrique définie positive alors la méthode de Gauss-Seidel converge
- La méthode de Gauss-Seidel qui est plus rapide que Jacobi. Jacobi est bien adaptée au calcul parallèle.
- Pour une méthode itérative, le cout d'une itération est faible dans le cas de matrices creuses.

La table de comparaison :
CP –> Ne converge pas

A[i]	p(j)	NbIter Jacobi	p(GS)	NbIter Gauss-Seidel
A1	0.0	2	CP	∞
A2	CP	∞	CP	∞
A3	0.000024	6	0.000031	5
A4	0.000113	40	0.000139	39
A5	0.001953	4	0.000244	4
A6	0.000181	7	0.000271	6

Table des figures