Prép'ISIMA

Programmation Linéaire

Implémentation de Gauss et Jacobi

Clément Payard
Mathieu Laurençot

Encadrant : M. Suzanne ÉLODIE

2021

Table des matières

1 Rappel rapide des méthodes

1.1 Méthode de Gauss

Cette méthode permet de trouver une solution exacte au système $Ax = b$ en un nombre fini d'étape.

Pour ce faire, cette méthode se fait en plusieurs étapes :

1. La triangularisation On doit passer du système $Ax = b$ au système $A'x = b'$ où A' est une matrice triangulaire supérieure. L'algorithme utilisé est disponible dans le programme.
2. La résolution facile `Nécessite aucun 0 sur la diagonale de A`

1.2 Méthode de Jacobi

Cette méthode fait partie des méthodes itératives, où l'on cherche à se rapprocher, avec une suite d'itération définie, à une solution exacte.

Pour cette méthode, nous devons tout d'abord décomposer A sous la forme A = D -E -F

1. D est la matrice nul de taille A, sauf sur sa diagonale où D possède les coefficients de A.
2. -E est la matrice triangulaire inférieure de A
3. -F est la matrice triangulaire supérieur de A

De plus, on pose $M = D$ et $N = E + F$

On obtient donc le système :

$$Ax = b \Longleftrightarrow Dx^{k+1} = (E + F)x^k + b$$

pour l'itération $k + 1$

De plus, l'algorithme de Jacobi s'écrit avec une précision ϵ :

2 Présentation des programmes commentés

2.1 Présentation Général :

2.1.1 Les différents fichiers utilisés

Pour effectuer ce travail, nous avons décidé de séparer notre programme en plusieurs fichiers :

1. main.c, qui est notre fichier appelant les divers fonctions présentent dans
2. fonction.c, puis
3. fonction.h, permettant de définir les différentes structures et les headers des fonctions, et enfin
4. main.h, où les différentes bibliothèques sont déclarées
5. De plus, il y a un Makefile, qui nous permet de compiler et tester notre programme efficacement

2.1.2 Les structures ainsi que les fonctions usuelles

1. La structure de nos matrice
   Nous avons choisis de définir notre structure matrice de la sorte :

```
  typedef struct matrice
{
  int longueur;
  int largeur;
  long double **Mat;
} matrice;
```

Mettre la longueur et la largeur de la matrice directement dans la structure permet à nos fonction de ne plus les avoir en paramètre. De plus, les matrices sont des doubles tableaux. On utilise dont un pointeur de pointeur pour permettre, grâce à la fonction

2. Les fonctions usuelles

(a) *creerMatrice*, qui prend un paramètre la largeur et la longueur, et qui renvoie un pointeur de matrice. Cette fonction fait principalement la création et l'initialisation d'un tableau 2D avec des 0.

```
matrice *creerMatrice(int largeur, int longueur)
{
  /* création d'un pointeur sur une structure matrice (avec l'espace
     alloué de la structure matrice)  */
  matrice *fini = (matrice *)malloc(sizeof(matrice));

  /* attribution de la longueur et de la largeur */
  fini->longueur = longueur;
  fini->largeur = largeur;

  /* Création du double tableau, la matrice même (initialisé à 0)  */
  fini->Mat = (long double **)malloc(fini->longueur * sizeof(long double *));
  /* parcourt de la matrice avec la double boucle for */
  for (int i = 0; i < fini->longueur; i++)
    {
      fini->Mat[i] = (long double *)malloc(fini->largeur * sizeof(long double)
      for (int j = 0; j < fini->largeur; j++)
{
  fini->Mat[i][j] = 0;
}
    }

  /* renvoie de la fini */
  return fini;
}
```

(b) *destroyMatrice*, qui permet de supprimer et de vider la mémoire d'une matrice.

```
void destroyMatrice(matrice *mat)
{
  for (int i = 0; i < mat->longueur; i++)
  {
    free(mat->Mat[i]);
  }
  free(mat->Mat);
  mat->longueur = 0;
  mat->largeur = 0;
  free(mat);
}
```

(c) / afficheMatrice/, qui permet tout simplement d'afficher une matrice.

(d) *remplisAleaBcpZero*, qui permet de remplir une matrice avec environ 70% de 0.

(e) *remplisAlea*, permettant de remplir une matrice avec des nombres aléatoire (etre -100 et 100)

(f) *remplisAleaInt*, déclinaison de remplisAlea avec des entier

(g) *additionMatrice*, qui, comme son nom l'indique, d'additionner 2 matrices

```
matrice *additionMatrice(matrice mat1, matrice mat2)
{
  // initialisation des variables
  matrice *fini;
  long double res;

  // initialisation de la matrice de retour
```

```
  // création de la matrice fini avec la taille de la taille max entre
  // 2 matrices
  fini = creerMatrice(
      (mat1.largeur > mat2.largeur) ? mat1.largeur : mat2.largeur,
      (mat1.longueur > mat2.longueur) ? mat1.longueur : mat2.longueur);

  // mise du resultat dans la matrice de retour
  for (int i = 0; i < fini->longueur; i++)
  {
    for (int j = 0; j < fini->largeur; j++)
    {
      res = 0;
      if ((mat1.longueur > i) && (mat1.largeur > j))
      {
res += mat1.Mat[i][j];
      }
      if ((mat2.longueur > i) && (mat2.largeur > j))
      {
res += mat2.Mat[i][j];
      }
      fini->Mat[i][j] = res;
    }
  }

  // retour de la matrice de resultat
  return fini;
}
```

(h) *soustractino*, qui permet de les soustraire

(i) *multiplicationMatrice*, qui les multiplie

```
matrice *multiplicationMatrice(matrice mat1, matrice mat2)
{
  // verification des conditions
  if (mat1.largeur != mat2.longueur)
  {
    printf("On ne peu pas multiplier ces deux matrices ensemble.\n");
    return NULL;
  }

  // initialisation des variables
  matrice *Xfini = creerMatrice(mat2.largeur, mat1.longueur);
  long double lambda;
  /* afficheMatrice(mat1); */
  // calcule de chaque case une par une
  for (int i = 0; i < mat1.longueur; i++)
  {
    for (int j = 0; j < mat2.largeur; j++)
    {
      lambda = 0;
      for (int h = 0; h < mat1.largeur; h++)
      {
lambda += (mat1.Mat[i][h] * mat2.Mat[h][j]);
      }
      Xfini->Mat[i][j] = lambda;
    }
  }

  return Xfini;
}
```

2.2 TODO Gauss

2.2.1 Gestion des 0 sur la diagonale

Tout d'abord, cette méthode requiert des nombres non nul sur la diagonale. Il faut donc tester, en échangeant certaines lignes, si il est possible d'obtenir une matrice avec aucun 0 sur les diagonales

```
int okayDiag = 0;


/* on compte le nombre de 0 sur la diagonale */
for (int i = 0; i < n; i++)
{
  if (res->Mat[i][i] == 0)
  {
    okayDiag++;
  }
}
while (okayDiag)
{
  int change = 0;
  for (int i = 0; i < n; i++)
  {
    if (res->Mat[i][i] == 0)
    {
      for (int j = 0; j < n; j++)
      {
if (j < i)
{
  if (res->Mat[i][j] != 0 && res->Mat[j][i] != 0)
  {
    swapLine(res, i, j);
    okayDiag--;
    change++;
    break;
  }
}
else if (j > i)
{
  if (res->Mat[j][i] != 0)
  {
    if (res->Mat[i][j] != 0)
    {
      okayDiag--;
    }
    swapLine(res, i, j);
    change++;
    break;
  }
}
      }
    }
  }
  if (change == 0)
  {
    okayDiag = 0;
    for (int i = 0; i < n; i++)
    {
      if (res->Mat[i][i] == 0)
      {
```

```
okayDiag++;
      }
    }
    if (okayDiag)
    {
      printf("La matrice ne peut avoir de diagonale sans zéros...\n");
      return res;
    }
  }
}
```

2.2.2 Application de la méthode

Puis, on peut appliquer la méthode de Gauss sur la matrice augmentée AB

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < i; j++)
  {
    lambda = -(A->Mat[i][j]);
    for (int k = 0; k < n; k++)
    {
      A->Mat[i][k] = lambda * A->Mat[j][k] + A->Mat[i][k];
    }
    res->Mat[i][0] = lambda * res->Mat[j][0] + res->Mat[i][0];
  }
  lambda = A->Mat[i][i];
  for (int k = 0; k < n; k++)
  {
    A->Mat[i][k] = A->Mat[i][k] / lambda;
  }
  res->Mat[i][0] = res->Mat[i][0] / lambda;
}
```

2.2.3 Application de calcul simple

Enfin, on peut appliquer le calcul simple sur matrice augmenté obtenue pour obtenir le résultat dans X

```
for (int i = n - 2; i > -1; i--)
  {
    lambda = 0;
    for (int j = i + 1; j < n; j++)
      {
lambda -= (A->Mat[i][j] * res->Mat[j][0]);
      }
    res->Mat[i][0] += lambda;
  }
```

2.3 Jacobi

2.3.1 Gestion des matrices non diagonales dominantes

Tout d'abord, pour utiliser la méthode de Jacobi, il faut que les matrices soient de la bonne taille. On vérifie donc que la matrice A est carré, que la longueur de A est égal à celle de B, ainsi que la largeur de B qui doit être égal à un. Ces conditions sont testés avec ce code :

```
/* gestion des cas d'erreur pouvant faire echouer la methode jacobi*/
if ((A->largeur != A->longueur) || (A->longueur != B->longueur) ||
    (B->largeur != 1))
{
  printf(
      "Les matrice ne sont pas de la taille nécessaire a leurs résolution.");
  return B;
}
 else
```

Puis, il faut aussi que les matrices soient strictement diagonales dominantes. Pour ce faire, nous pouvons mettre au début de la fonction "Jacobi", une double boucle for qui test si les diagonales sont bien dominantes :

```
  else
{
  for (int i = 0; i < A->longueur; i++)
  {
    int verifieur = 0;
    for (int j = 0; j < A->longueur; j++)
    {
      if (j != i)
      {
verifieur += fabsl(A->Mat[i][j]);
      }
    }
    if (verifieur > A->Mat[i][i])
    {
      printf("La matrice n'est pas à diagonale dominante et ne vas donc pas "
     "converger...\n");
      return B;
    }
  }
}
```

Sinon, le reste du code est exécuté normalement.

2.3.2 Méthode général

1. Initialisation des différentes matrices nécessaires
   Suites à la création des matrices x, D, E, F (et N) à l'aide des fonction usuelles, il faut les initialiser avec les bonnes valeurs. On a choisit de faire un parcours de la matrice A, et lorsque les conditions sont réunies, nous entrons la valeur de A en fonction de la matrice.

   ```
     /* initialisation de D E et F (et N)*/
   for (int i = 0; i < A->longueur; i++)
   {
     for (int j = 0; j < A->largeur; j++)
   ```

```
  {
    if (i == j)
    {
      D->Mat[i][j] = A->Mat[i][j];
      E->Mat[i][j] = 0;
      F->Mat[i][j] = 0;
    }
    else if (i < j)
    {
      D->Mat[i][j] = 0;
      E->Mat[i][j] = -(A->Mat[i][j]);
      F->Mat[i][j] = 0;
    }
    else
    {
      D->Mat[i][j] = 0;
      E->Mat[i][j] = 0;
      F->Mat[i][j] = -(A->Mat[i][j]);
    }
    N->Mat[i][j] = E->Mat[i][j] + F->Mat[i][j];
  }
}
```

2. Inversion de la matrice D
   Puis, on initialise la marge d'erreur. Nous inversons également D (avec la fonction *InversematriceD*), car il faut utiliser D^{-1}

```
void InversematriceD(int taille, matrice *D)
{
  for (int i = 0; i < D->longueur; i++)
    {
      for (int j = 0; j < D->largeur; j++)
{
  if ((i == j) && (D->Mat[i][j] != 0))
    {
      D->Mat[i][j] = 1 / D->Mat[i][j];
    }
}
    }
}
```

```
float erreur = Eps + 1;
InversematriceD(D->longueur, D);
```

3. Méthode de Jacobi appliquée Enfin, nous programmons la méthode de Jacobi grâce aux fonctions "multiplication-Matrice, additionMatrice". La boucle utiliser est un tant que, car nous ne savons pas quand la marge d'erreur sera respecté pour sortir de la boucle while. La variable *erreur* doit également être mise à jour à chaque passage dans la boucle, en utilisant la fonction *Norme*.

```
float Norme(matrice *colonne)
{
  float norme = 0;
  for (int i = 0; i < colonne->longueur; ++i)
    {
      norme = norme + pow(colonne->Mat[i][0], 2);
```

```
    }
  return sqrt(norme);
}

while (erreur > Eps)
  {
    // nouvelle valeur de x selon la formule
    x = multiplicationMatrice(*D, *additionMatrice(*(multiplicationMatrice(*N, *
    // nouvelle valeur d'erreur selon la formule
    erreur = Norme(soustractino(*multiplicationMatrice(*A, *x), *B));

  }
return x;
}
```

2.4 Programme final

Les différentes fonctions sont appelées au fur et à mesure du main.c, en laissant le choix à l'utilisateur des matrices qu'il veut testes, ainsi que la méthode à utiliser pour la résolution. Les différents choix sont regroupés dans un switch.

3 Présentation des matrices test au dos de la feuille

Nous ne mettrons pas toutes les matrices test, mais vous pouvez les retrouver ici.

De plus, les matrices test sont compatibles seulement avec Gauss. En effet, seule la matrice $Km_{carré}$ avec un p petit (<= 0.3) sera dominantes.

Pour tester les matrices test, il suffit de donner à A une taille n*n, puis de la remplir grâce au programme de la matrice test voulue. Le temps d'exécution de la résolution sera ainsi donné, avec la solution X.

4 TODO Commentaires des jeux d'essais à partir de données relatives [0/4]

Donc pour résumer : -% d'écart = diff entre B' calculé et B donné -font° d'erreur = norme de Ax - b -vitesse de convergence = temps d'execution ?

complexité pratique= O(n)

vitesse de convergence = nombre d'itération (que sur jacobi)

4.1 NEXT Pourcentage d'écart (précision)

4.2 TODO Calcul de fonction d'erreurs

4.3 TODO Vitesse de convergence

4.4 TODO Complexité pratique (plus qu'à mettre)

5 Conclusion sur les méthodes

5.1 Comparaison

Comme on peut le voir sur les différents jeux d'essais, Jacobi est plus efficace que Gauss, car sa complixité est $3n^2+2n$ par itération, et donc d'ordre $O(n^2)$, tandis que Gauss a une complixité égale à $2\frac{n^3}{3}$, car $\frac{n^2}{2}$ pour la division, et $\frac{n^3}{3}$ pour la multiplication. Gauss a donc une complexité d'ordre $O(n^3)$. Malgré tout, il existe des cas où la méthode de Jacobi prendra plus de temps : Il suffit de donner un ϵ trés petit, et donc d'augmenter la précision du résultat voulu, pour entraîner un nombre d'itération trop conséquent pour effectuer le calcul.

5.2 TODO Cadre d'utilisation

5.3 TODO Stabilité