Figure 1: logo uca

TP

Algorithmique Numérique

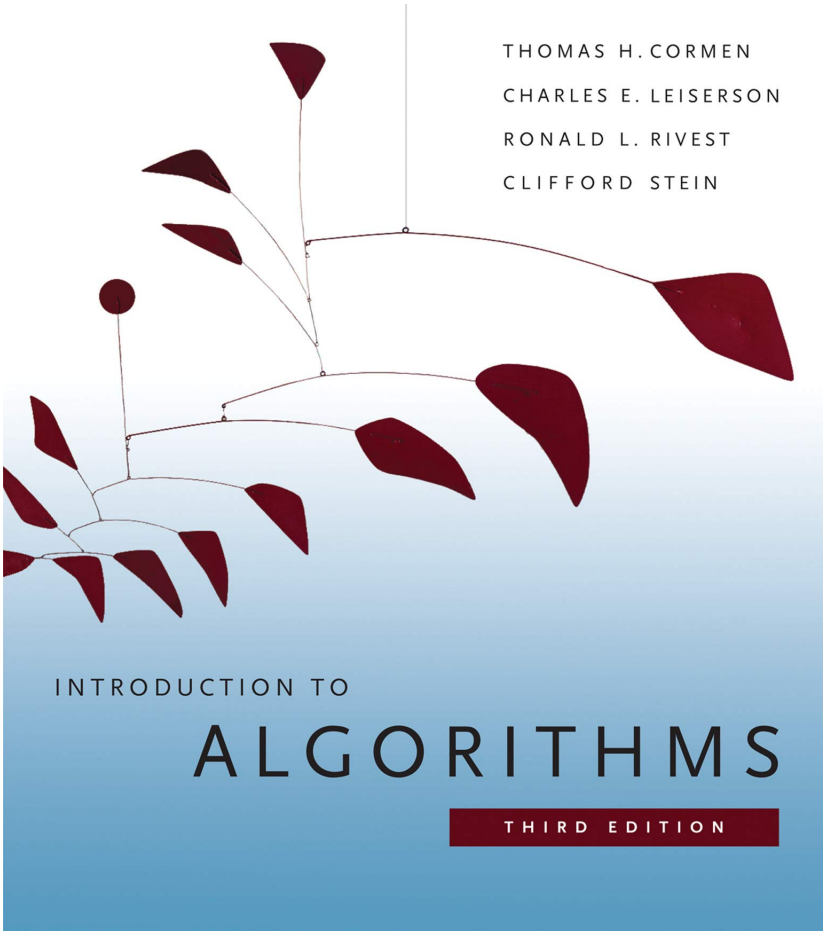

Figure 2: Cormen

SABIER Corentin et CHASSAGNOL Rémi
année 2020-2021

Professeur: CHORFI Amina

24 octobre 2020

Table des matières

1 Algorithmiques directes

1.1 Gauss

1.2 Cholesky

1.2.1 Principe

Le principe de cette méthode est de trouver une matrice R telles que $A = R^T R$ où R est une matrice triangulaire supérieure et R^T est la matrice transposée de R. Le système $Ax = b$ peut donc se diviser en deux sous-systèmes $R^T y = b$ et $Rx = y$. Cette méthode est légèrement plus rapide que celle de Gauss avec une complexité de $\frac{n^3}{3}$ au lieu de $\frac{2n^3}{3}$, cependant elle nécessite que la matrice soit définie positive.

1.2.2 L'Algorithme

L'algorithme qui permet de trouver la matrice R est le suivant:

```
Pour i allant de 1 a n:
    s = a_ii − Σ_{j=1}^{i−1} r_ji^2
    Si s ⩽ 0 alors:
        arret("la matrice n'est pas definie positive")
    Sinon
        r_ii = √s
        Pour j allant de i+1 a n:
            r_ij = (a_ij − Σ_{k=1}^{i−1}) / r_ii
```

Ensuite il suffit de résoudre les deux sous-systèmes ce qui ce fait facilement car la matrice est triangulaire.

1.2.3 Les Variables

Dans la fonction *main* on initialise un matrice de taille **LEN** qui est une constante pré-processeur. Les valeurs dans la matrice dépendent de la constante **EXAMPLE** qui permet de déterminer l'initialisation de la matrice via une fonction qui se situe dans le fichier *matrix.c*. Il y a en tout 8 matrices de testes qui sont celle demandé dans la fiche de TP (chaque fonction remplit le tableau en fonction de la définition de la matrice).

```c
float *solus = malloc(LEN * sizeof(float));
if (solus == NULL)
  exit(EXIT_FAILURE);
float *b = malloc(LEN * sizeof(float));
if (b == NULL)
  exit(EXIT_FAILURE);
for (i = 0; i < LEN; i++) {
  b[i] = 1;
}
float **matrix = create_mat(LEN);
mat(matrix, LEN, EXAMPLE);
show(matrix, LEN);
solver_cholesky(matrix, solus, b, LEN);
```

1.2.4 Calcule de la matrice R

Ensuite on calcule la matrice R (le tableau à deux dimensions r) à l'aide de la fonction nommée **make_R** qui suit l'algorithme ci-dessus. A noter que cette fonction renvoi une erreur si la matrice en entrée n'est pas définie positive ce qui stop le programme car le système n'est pas résolvable avec cette méthode (en fait si la matrice n'est pas définie positive, la matrice R n'existe pas).

```c
void make_R(float **matrix, float **r, int n) {
  int i, j, k;   // loop var
  float sum = 0; // Used to calculate the sum of r_ki*r_kj
  float s;
```

```
  for (i = 0; i < n; i++) {
    s = matrix[i][i];
    for (j = 0; j < i; j++) {
      s -= r[j][i] * r[j][i];
    }
    if (s <= 0) {
      printf("La matrice n'est pas d finie positive\n");
      exit(EXIT_FAILURE);
    } else {
      r[i][i] = sqrtf(s);
      for (j = (i + 1); j < n; j++) {
        sum = 0;
        for (k = 0; k < i; k++) {
          sum += r[k][i] * r[k][j];
        }
        r[i][j] = (matrix[i][j] - sum) / r[i][i];
      }
    }
  }
}
```

1.2.5 Résolution des sous-systèmes

Enfin la fonction **solver_cholesky** résout les deux sous-systèmes de la même manière qu'avec *Gauss* sauf que les a_{ii} ne sont pas forcement égaux à 1 d'où la division pas a_{ii}. De plus, dans le premier sous-système, la matrice R^T est triangulaire inférieure, la résolution se fait donc en commencent par calculer la valeur de la première inconnue.

```
void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
  int i, j;
  float tmp;
  float *solus_y = malloc(n * sizeof(float));
  /*Creation of the matrix  R:*/
  float **r = create_mat(n);
  mat_0(r, n);
  make_R(matrix, r, n);
  /*Resolution de R^T * y = b*/
  transpose(r, n);
  for (i = 0; i < n; i++) {
    tmp = b[i];
    for (j = 0; j < i; j++) {
      tmp -= solus_y[j] * r[i][j];
    }
    solus_y[i] = tmp / r[i][i];
  }
  /*Resolution R * x = y*/
  transpose(r, n);
  for (i = (n - 1); i >= 0; i--) {
    for (j = (n - 1); j > i; j--) {
      solus_y[i] -= solus_x[j] * r[i][j];
    }
    solus_x[i] = solus_y[i] / r[i][i];
  }
  /*FREE*/
  free(solus_y);
  for (i = 0; i < n; i++) {
    free(r[i]);
  }
  free(r);
}
```

Cette fonction fait appelle à plusieurs autres fonctions:

- **create_mat** qui retourne un tableau à deux dimensions de taille n
- **mat_0** qui remplit un tableau à deux dimensions de 0
- **make_R** vue ci dessus

- **transpose** qui transpose la matrice qui lui est donnée en argument

Observons plus en détail la résolution du premier sous-système:

```
1    for (i = 0; i < n; i++) {
2      for (j = 0; j < i; j++) {
3        b[i] -= solus_y[j] * r[i][j];
4      }
5      solus_y[i] = b[i] / r[i][i];
6    }
```

Voici la matrice r (après avoir été transposé):

$$\begin{vmatrix} a_{11} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \vdots & \cdots & \ddots & 0 & 0 \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{vmatrix}$$

Nous résolvons $Ry = b$ donc:

$$y_i = \frac{b_i - \sum_{j=1}^{i}}{a_{ii}}$$

Ce qui se fait à l'aide des deux boucles ci-dessus.

A noter que ce programme alloue plus de mémoire que le précédent à cause de l'utilisation du tableau **r** ce qui pourrait être problématique si on travail avec des matrices de grandes taille.

1.2.6 Les Tests

- Bord

```
| 1.000000 | 0.500000 | 0.250000 | 0.125000 |
| 0.000000 | 1.000000 | 0.000000 | 0.000000 |
| 0.000000 | 0.000000 | 1.000000 | 0.000000 |
| 0.000000 | 0.000000 | 0.000000 | 1.000000 |
0.186047
0.906977
0.953488
0.976744
```

Cette solution est très imprécise avec cette méthode ce qui est du au fait que le programme crée la matrice R et résous les deux sous-systèmes avec une matrice qui est déjà triangulaire supérieure ce qui génère une suite d'imprécisions dans les calcule. la vrai solution est: $x_1 = \frac{1}{8} = 0.125, x_2 = 1, x_3 = 1, x_4 = 1$

- Ding Dong

```
| 1.750000 | 1.250000 | 0.750000 | 0.250000 |
| 1.250000 | 0.750000 | 0.250000 | -0.250000 |
| 0.750000 | 0.250000 | -0.250000 | -0.750000 |
| 0.250000 | -0.250000 | -0.750000 | -1.250000 |
La matrice n'est pas définie positive
```

Les programme retourne une erreur, en effet, la matrice n'est pas définie positive donc la méthode ne converge pas.

- Franc

```
| 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| 1.000000 | 2.000000 | 2.000000 | 2.000000 |
| 0.000000 | 2.000000 | 3.000000 | 3.000000 |
| 0.000000 | 0.000000 | 3.000000 | 4.000000 |
1.000000
0.000000
0.000000
0.000000
```

Le programme trouve la bonne solution.

1.3 Conclusion sur les méthodes de résolution directes

j'aime les chats!!!

2 algorithmes itératifs

Les algorithmes itératifs sont des algorithmes qui vont trouver une solution de plus en plus précise à chaque itération. Ils partent d'un point de départ arbitraire (un vecteur arbitraire) et vont converger vers la solution du système.

2.1 Jacobi

2.1.1 Principe

Cet algorithme détermine les solutions du système $Ax = b$ en utilisant la formule suivante:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}}$$

A chaque tours de boucle, l'algorithme calcule le nouveau vecteur $x^{(k+1)}$ en utilisant la solution $x^{(k)}$ qu'il a calculé à l'itération précédente qui au premier tour est un vecteur choisit arbitrairement qui sert de point de départ. L'algorithme s'arrête quand la solution trouvée est convenable $\| x^{(k+1)} - x^{(k)} \| \leqslant \varepsilon$ (ou ε est la précision choisit) ou quand il a fait un nombre de tours pré-déterminer (pour éviter les boucles infinie si la méthode ne converge pas).

2.1.2 L'algorithme

---

1 Tant que $\varepsilon^{(k)} \geqslant \varepsilon$:
2 $x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}}$
3 $\varepsilon^{(k+1)} = \| x^{(k+1)} - x^{(k)} \|$

2.1.3 Implémentation en C

Les variables sont initialisées de la même façon que dans les méthodes directes vues précédemment et le point de départ est **solus_k** qui vaut $(1, \ldots, 1)$. La fonction qui calcule les solutions est la suivante:

```c
void jacobi(float **matrix, float *b, float *solus_k, int n) {
  int i, j;
  int compt = 0;
  int bool = 1;
  float *solus_k1 = malloc(n * sizeof(float));
  while (bool && compt < MAX) {
    for (i = 0; i < n; i++) {
      solus_k1[i] = b[i];
      for (j = 0; j < n; j++) {
        if (j != i)
          solus_k1[i] -= matrix[i][j] * solus_k[j];
      }
      solus_k1[i] /= matrix[i][i];
    }
    bool = test(solus_k1, solus_k, n);
    for (i = 0; i < n; i++) {
      solus_k[i] = solus_k1[i];
    }
    compt++;
  }
  printf("Nb tours: %d\n", compt);
  // FREE
  free(solus_k1);
}
```

Cette fonction utilise le même principe que l'algorithme ci dessus, elle fait appel à une fonction *test* qui retourne $\| x^{(k+1)} - x^{(k)} \| \leqslant \varepsilon$ qui détermine l'arrêt (où sinon la fonction s'arrête d'itérer à un nombre de tours maximum). A la fin =, on affiche le nombre de tours effectués pour avoir un idée de la vitesse de convergence.

2.2 Gauss-seidel

2.2.1 Le principe

Cette méthode repose sur un principe similaire à celle de *Jacobi*, cependant, elle utilises les solutions $x^{(k+1)}$ au fur et à mesure qu'elle les calcules, ce qui lui permet de converger plus vite et donc de trouver une solution plus rapidement. La formule utilisée par la méthode est la suivante:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(\mathbf{k+1})} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}}$$

Cette méthode converge lorsque la matrice est définie positive et symétrique, ou si elle est à diagonale dominante.

2.2.2 L'algorithme

L'algorithme est le même qu'avec **Jacobi** sauf qu'il utilise la nouvelle formule:

---

1 Tant que $\varepsilon^{(k)} \geqslant \varepsilon$:
2 $x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}}$
3 $\varepsilon^{(k+1)} = \| x^{(k+1)} - x^{(k)} \|$

2.2.3 Le programme en C

la fonction utilisée dans ce programme est très similaire à celle utilisée avec **Jacobi** mis à part qu'elle utilise l'autre formule ce qui ce traduit part un changement de variable.

```
1  void gauss_seidel(float **matrix, float *b, float *solus_k, int n) {
2    int i, j;
3    int compt = 0;
4    int bool = 1;
5    float *solus_k1 = malloc(n * sizeof(float));
6    while (bool && compt < MAX) {
7      for (i = 0; i < n; i++) {
8        solus_k1[i] = b[i];
9        for (j = 0; j < i; j++) {
10         solus_k1[i] -= matrix[i][j] * solus_k1[j];
11       }
12       for (j = i+1; j < n; j++) {
13     solus_k1[i] -= matrix[i][j] * solus_k[j];
14       }
15       solus_k1[i] /= matrix[i][i];
16     }
17     bool = test(solus_k1, solus_k, n);
18     for (i = 0; i < n; i++) {
19       solus_k[i] = solus_k1[i];
20     }
21     compt++;
22   }
23   printf("Nb tours: %d\n", compt);
24   // FREE
25   free(solus_k1);
26 }
```

La modification est au niveau de la ligne **10**, on utilise $x^{(K+1)}$ au lieu de $x^{(k)}$

2.3 Comparaison des résultats

2.3.1 Matrice symétrique à diagonale dominante

Dans cette exemple nous allons résoudre deux systèmes avec la même matrice symétrique à diagonale supérieure:

Résolution de:

$$\begin{pmatrix} 4 & 1 & 1 & 0 \\ 1 & 4 & 0 & 1 \\ 1 & 0 & 4 & 1 \\ 0 & 1 & 1 & 4 \end{pmatrix} X = \begin{pmatrix} 15 \\ 15 \\ 19 \\ 11 \end{pmatrix}$$

Voici la solution trouvée par **jacobi**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 24
2.000000
3.000000
4.000000
1.000000
```

Voici la solution trouvée par **gauss seidel**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 13
2.000000
3.000000
4.000000
1.000000
```

On constate que la solution trouvée avec les deux méthodes et la même et si on regarde le nombre de tours effectués, on voit que **gauss-seidel** converge presque deux fois plus vite que **jacobi**.

Résolvons le même système en faisant varier les valeurs dans b de +0.005:

Voici la solution trouvée par **jacobi**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 25
2.000834
3.000833
4.000833
1.000833
```

Voici la solution trouvée par **gauss seidel**:

```
| 4.000000 | 1.000000 | 1.000000 | 0.000000 |
| 1.000000 | 4.000000 | 0.000000 | 1.000000 |
| 1.000000 | 0.000000 | 4.000000 | 1.000000 |
| 0.000000 | 1.000000 | 1.000000 | 4.000000 |
Nb tours: 13
2.000834
3.000833
4.000833
1.000833
```

On constate que les résultats n'ont pas beaucoup variés, donc la modification effectuée sur b sont bien gérées par le programme donc les méthodes sont stables.

2.3.2 Une Erreur

Avec ce test, on voit que si la matrice ne respecte pas les contrainte requises par les méthodes, celle-ci tournes à l'infini sans jamais converger vers la solution du système (le programme s'arrête car la fonction est limité à 100 tours).

Résolution de:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 2 & 3 \\ 5 & 1 & 8 \end{pmatrix} X = \begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix}$$

Voici la solution trouvée par **jacobi**:

```
| 1.000000 | 2.000000 | 1.000000 |
| 2.000000 | 2.000000 | 3.000000 |
| 5.000000 | 1.000000 | 8.000000 |
Nb tours: 100
1566439752197329627469756497 92.000000
1243872464955820438002377687 04.000000
5794099055897242029582922547 2.000000
```

Voici la solution trouvée par **gauss seidel**:

```
| 1.000000 | 2.000000 | 1.000000 |
| 2.000000 | 2.000000 | 3.000000 |
| 5.000000 | 1.000000 | 8.000000 |
Nb tours: 100
-18743254056960.000000
-181350950764544.000000
34383402631168.000000
```