Figure 1: logo uca

TP

Algorithmique Numérique

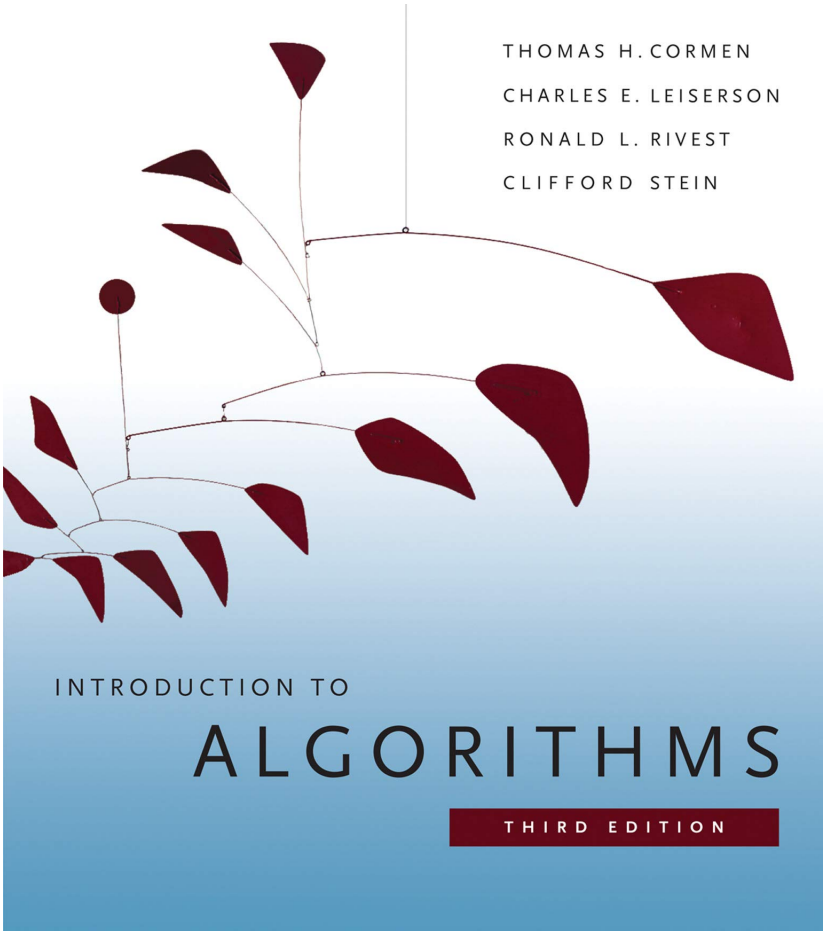

Figure 2: Cormen

SABIER Corentin et CHASSAGNOL Rémi
année 2020-2021

Professeur: CHORFI Amina

24 octobre 2020

Table des matières

1 Algorithmiques directes **2**
1.1 Gauss . . . 2
1.2 Cholesky . . . 2
1.2.1 Principe . . . 2
1.2.2 L'Algorithme . . . 2
1.2.3 Les Variables . . . 2
1.2.4 Calcule de la matrice R . . . 2
1.2.5 Résolution des sous-systèmes . . . 3
1.2.6 Les Tests . . . 4
1.3 Conclusion sur les méthodes de résolution directes . . . 5

2 algorithmes itératifs **5**
2.1 Jacobi . . . 5
2.1.1 Principe . . . 5
2.1.2 L'algorithme . . . 6
2.1.3 Implémentation en C . . . 6
2.2 Gauss-seidel . . . 6

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:

```
1  Pour i allant de 1 a n:
2      s = a_ii − Σ_{j=1}^{i−1} r_ji^2
3      Si s ⩽ 0 alors:
4          arrt(la matrice n'est pas definie positive)
5      Sinon
6          r_ii = √s
7          Pour j allant de i+1 a n:
8              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).

```
1   float *solus = malloc(LEN * sizeof(float));
2   if (solus == NULL)
3     exit(EXIT_FAILURE);
4   float *b = malloc(LEN * sizeof(float));
5   if (b == NULL)
6     exit(EXIT_FAILURE);
7   for (i = 0; i < LEN; i++) {
8     b[i] = 1;
9   }
10  float **matrix = create_mat(LEN);
11  mat(matrix, LEN, EXAMPLE);
12  show(matrix, LEN);
13  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).

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

```
5     for (i = 0; i < n; i++) {
6       s = matrix[i][i];
7       for (j = 0; j < i; j++) {
8         s -= r[j][i] * r[j][i];
9       }
10      if (s <= 0) {
11        printf("La matrice n'est pas d finie positive\n");
12        exit(EXIT_FAILURE);
13      } else {
14        r[i][i] = sqrtf(s);
15        for (j = (i + 1); j < n; j++) {
16          sum = 0;
17          for (k = 0; k < i; k++) {
18            sum += r[k][i] * r[k][j];
19          }
20          r[i][j] = (matrix[i][j] - sum) / r[i][i];
21        }
22      }
23    }
24  }
```

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.

```
1   void solver_cholesky(float **matrix, float *solus_x, float *b, int n) {
2     int i, j;
3     float *solus_y = malloc(n * sizeof(float));
4     /*Cr ation de  R:*/
5     float **r = create_mat(n);
6     mat_0(r, n);
7     make_R(matrix, r, n);
8     /*Resolution de R^T * y = b*/
9     transpose(r, n);
10    for (i = 0; i < n; i++) {
11      for (j = 0; j < i; j++) {
12        b[i] -= solus_y[j] * r[i][j];
13      }
14      solus_y[i] = b[i] / r[i][i];
15    }
16    /*R solution de R * x = y*/
17    transpose(r, n);
18    for (i = (n - 1); i >= 0; i--) {
19      for (j = (n - 1); j > i; j--) {
20        solus_y[i] -= solus_x[j] * r[i][j];
21      }
22      solus_x[i] = solus_y[i] / r[i][i];
23    }
24    /*FREE*/
25    free(solus_y);
26    for (i = 0; i < n; i++) {
27      free(r[i]);
28    }
29    free(r);
30  }
```

A noter que cette fonction a pour effet de bord de modifier le vecteur **b** qui contient au départ les solution. Ce n'est pas problématique dans ce cas car **b** n'est pas réutilisé dans la suite du programme, en revanche si c'était le cas, il faudrait utiliser une variable tampon.

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.

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
```

- 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
```

- 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
```

- Hilbert -

```
| 1.000000 | 0.500000 | 0.333333 | 0.250000 |
| 0.500000 | 0.333333 | 0.250000 | 0.200000 |
| 0.333333 | 0.250000 | 0.200000 | 0.166667 |
| 0.250000 | 0.200000 | 0.166667 | 0.142857 |
-3.999544
59.994968
-179.987930
139.992157
```

- Hilbert +

```
| 0.333333 | 0.250000 | 0.200000 | 0.166667 |
| 0.250000 | 0.200000 | 0.166667 | 0.142857 |
| 0.200000 | 0.166667 | 0.142857 | 0.125000 |
| 0.166667 | 0.142857 | 0.125000 | 0.111111 |
-59.896725
419.399963
-838.973511
503.461731
```

- kms

```
| 1.000000 | 2.000000 | 4.000000 | 8.000000 |
| 0.500000 | 1.000000 | 2.000000 | 4.000000 |
| 0.250000 | 0.500000 | 1.000000 | 2.000000 |
| 0.125000 | 0.250000 | 0.500000 | 1.000000 |
La matrice n'est pas définie positive
```

- Lehmer

```
| 1.000000 | 0.500000 | 0.333333 | 0.250000 |
| 0.500000 | 1.000000 | 0.666667 | 0.500000 |
| 0.333333 | 0.666667 | 1.000000 | 0.750000 |
| 0.250000 | 0.500000 | 0.750000 | 1.000000 |
0.666667
0.266667
0.171429
0.571429
```

- Lotkin

```
| 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| 0.500000 | 0.333333 | 0.250000 | 0.200000 |
| 0.333333 | 0.250000 | 0.200000 | 0.166667 |
| 0.250000 | 0.200000 | 0.166667 | 0.142857 |
La matrice n'est pas définie positive
```

- Moler

```
| 1.000000 | -1.000000 | -1.000000 | -1.000000 |
| -1.000000 | 2.000000 | 0.000000 | 0.000000 |
| -1.000000 | 0.000000 | 3.000000 | 1.000000 |
| -1.000000 | 0.000000 | 1.000000 | 4.000000 |
43.000000
22.000000
12.000000
8.000000
```

1.3 Conclusion sur les méthodes de résolution directes

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 (ou le vecteur arbitraire pour le premier tour). 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

Tant que $\varepsilon^{(k)} \geqslant \varepsilon$:

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

$\varepsilon^{(k+1)} = \| x^{(k+1)} - x^{(k)} \|$

2.1.3 Implémentation en C

Les variables sont initialisées de la même façon que dans les méthodes directes vues précédemment. 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).

2.2 Gauss-seidel