

Figure 1: logo uca

## PRÉP'ISIMA

ADRESSE: campus Cézeaux, Clermont-Ferrand  
INTERNET: [www.uca.fr](http://www.uca.fr) et [www.isima.fr](http://www.isima.fr)

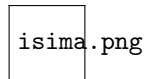


Figure 2: logo ISIMA

# Algorithmique Numérique

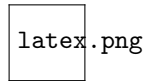


Figure 3: logo

SABIER Corentin et CHASSAGNOL Rémi  
année 2020-2021

*Professeur:* CHORFI amina

24 OCTOBRE 2020

## Table des matières

0.0.1	Principe . . . . .	2
0.0.2	L'Algorithme . . . . .	2
0.0.3	Implémentation en C . . . . .	2

## Algorithmiques directes

### Gauss

### Cholesky

#### 0.0.1 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.

#### 0.0.2 L'Algorithme

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

---

```

1  Pour  $i$  allant de 1 à  $n$ :
2       $s = a_{ii} - \sum_{j=1}^{i-1} r_{ji}^2$ 
3      Si  $s \leq 0$  alors:
4          arr t (la matrice n'est pas d finie positive)
5      Sinon
6           $r_{ii} = \sqrt{s}$ 
7          Pour  $j$  allant de  $i+1$  à  $n$ :
8               $r_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj}}{r_{ii}}$ 
```

---

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

#### 0.0.3 Implémentation en C

Dans la fonction *main* on initialise une 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 tests qui sont celle demandée 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);
```

---

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.

---

```

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 }

```

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 forcément égaux à 1 d'où la division par  $a_{ii}$ . De plus, dans le premier sous-système la matrice  $R^T$  est triangulaire inférieure, la résolution se fait donc en commençant 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     /* Creation of the matrix 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    /* Solving 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 }

```

## algorithmes itératifs

### Jacobi

### Gauss-seidel