INFORMATIQUE 4: ALGORITHMES NUMÉRIQUES

Jacques Tagoudjeu

Department MSP
Ecole Nationale Supérieure Polytechnique,
Université de Yaounde I

March 19, 2020

Outline

Objectifs

- Introduction à l'élaboration et l'étude des algorithmes pour la résolution des problèmes Mathématiques;
- Implémentation des algorithmes à l'aide d'un outil de calcul scientifique (Matlab/Octave, Python, Scilab, etc).

Ces objectifs s'inscrivent bien dans la culture scientifique:

- Justifier soigneusement la démarche;
- Bien choisir la méthode, si possible efficace;
- Lors des calculs numériques, contrôler les erreurs;
- Interpréter prudemment les résultats;
- Vérifier, recouper, utiliser le bon sens.

Contenu du Cours

1. Rappels d'algorithmique (Structure, Algorithmes Récursif)
2. Complexités des algorithmes;
3. Notions d'erreur, Représentation des Nombres, calcul exact vs calcul approché;
4. Introduction à Matlab;
5. Sommation numériques (Suites et Séries Numériques); Implementation des fonctions usuelles (Développement de Taylor, séries entières);
6. Arithmétique des polynômes;
7. Calcul matriciel, systèmes d'équations linéaires;
8. Interpolation de Lagrange, approximation polynomiale;
9. Intégration numérique : méthodes de rectangles, trapèzes, Simpson, méthodes stochastiques;
10. Méthodes itératives : méthode du point fixe, méthode de Newton;

Références du Cours

1. Algorithmique de base, *Support de Cours*.
   Auteur: **Dr NDONG NGUEMA**;
2. Sommation Numérique des séries, *Support de Cours*.
   Auteur: **Dr NDONG NGUEMA**;
3. Introduction à MATLAB et GNU Octave.
   Auteur: **Jean Daniel Bonjour**;
   disponible en ligne à $http://enacit1.epfl.ch/cours\_matlab/$ (version html) ou $http://enacit1.epfl.ch/cours\_matlab/pdf/Matlab-Octave-Cours-JDBonjour-2013-09-17.pdf$ (version pdf);
4. Introduction to Algorithm, 3rd ed., T. H. Cormen, C. E. LEISERSON, R. L. RIVEST, C. STEIN, MIT Press, 2009
5. Programming for Computations -MATLAB/Octave, S. Linge, H. P. Langtangen, Springer, 2016
6. Programming for Computations -PYTHON, S. Linge, H. P. Langtangen, Springer, 2016

Outline

Langage d'Expression Algorithmique (LEA)

- L'ordinateur présente l'avantage d'exécuter beaucoup de traitements complexes plus vite et plus sûrement que l'homme.
- Cependant il ne peut exécuter que les ordres qu'on lui donne à travers un programme informatique qui doit être établi de manière à envisager toutes les éventualités possibles.
- Un programme informatique (ou code) est généralement écrit dans un langage de programmation, que l'ordinateur interprète et exécute afin de fournir les résultats escomptés.

Langage d'Expression Algorithmique (Suite)

- Il existe un nombre élevé de langage de programmation, présentant chacun ses spécificités (orienté à résoudre une classe de problèmes).
- Il est nécessaire d'adopter une base universelle commune, compréhensible par le plus grand nombre, sur laquelle s'appuyer pour concevoir et écrire initialement les programmes informatiques, qui pourront être traduits dans des langages de programmation appropriés.
- Pour cela, on utilise un **Langage d'Expression Algorithmique (LEA) ou un pseudo-code**.

Étapes d'un algorithme

- **Définir clairement le problème**: Déterminer toutes les informations disponibles sur les données et la forme des résultats attendus.
  - Spécification d'un ensemble de données (énoncé, hypothèses, sources externes, ...);
  - Spécification d'un ensemble de buts à atteindre (résultats, opérations à effectuer, ...);
  - Spécification des contraintes.
- **Analyser le problème**: Rechercher une méthode de résolution en déterminant un moyen pour passer des données aux résultats.
  - Clarifier l'énoncé;
  - Simplifier le problème;
  - S'assurer que le problème est soluble;
  - Recherche d'une stratégie de construction de l'algorithme;
  - Décomposer le problème en sous problèmes partiels plus simples : raffinement;
  - Effectuer des raffinements successifs (Le niveau de raffinement le plus élémentaire est celui des instructions de base).

Étapes d'un algorithme (suite)

- **Écrire l'algorithme**: Décrire de manière détaillée les différentes étapes pour passer des données aux résultats. On procède par raffinement successif pour des algorithmes *complexes*
  - Les types des données et des résultats doivent être précisés;
  - L'algorithme doit fournir au moins un résultat;
  - L'algorithme doit être exécuté en un nombre fini d'opérations;
  - L'algorithme doit être spécifié clairement, sans la moindre ambiguïté.

Quelques définitions du mot algorithme

Définitions:

1. Un algorithme est le résultat d'une démarche logique de résolution d'un problème.
2. C'est une série d'actions ou d'opérations élémentaires qu'il faut exécuter en séquence pour accomplir une tâche quelconque, en suivant un enchaînement strict.
3. C'est une procédure de calcul bien définie qui prend en entrée une ou plusieurs valeurs appelées **données** et produit en sortie une ou un ensemble de valeurs (ou actions) appelées **résultats**.
4. C'est une suite d'étapes de calcul permettant de transformer les données en résultats.
5. C'est un ensemble de règles permettant d'effectuer un calcul soit à la main soit par la machine.
6. C'est un enchaînement d'actions nécessaires à l'accomplissement d'une tâche.

Propriétés d'un algorithme

Propriétés:

1. **Généralité**: Un algorithme doit être construit de manière à envisager toutes les éventualités.
2. **Finitude**: Il doit être fini et doit se terminer après un nombre fini d'opérations.
3. **Définitude**: Chaque étape de l'algorithme doit être clairement définie et toutes les opérations doivent être définies sans ambiguïté.
4. S'il y a des données, d'entrée, le domaine d'application doit être préciser (les Types des données).
5. Il doit posséder au moins un résultat.
6. **Effectivité**: Toutes les opérations doivent pouvoir être exécutées exactement et en temps fini.
7. **Efficacité**: Idéalement, un algorithme doit être conçu de telle sorte qu'il se déroule en un temps minimal et qu'il consomme un minimum de ressources.

Les composants du LEA

Objets et Éléments Algorithmiques

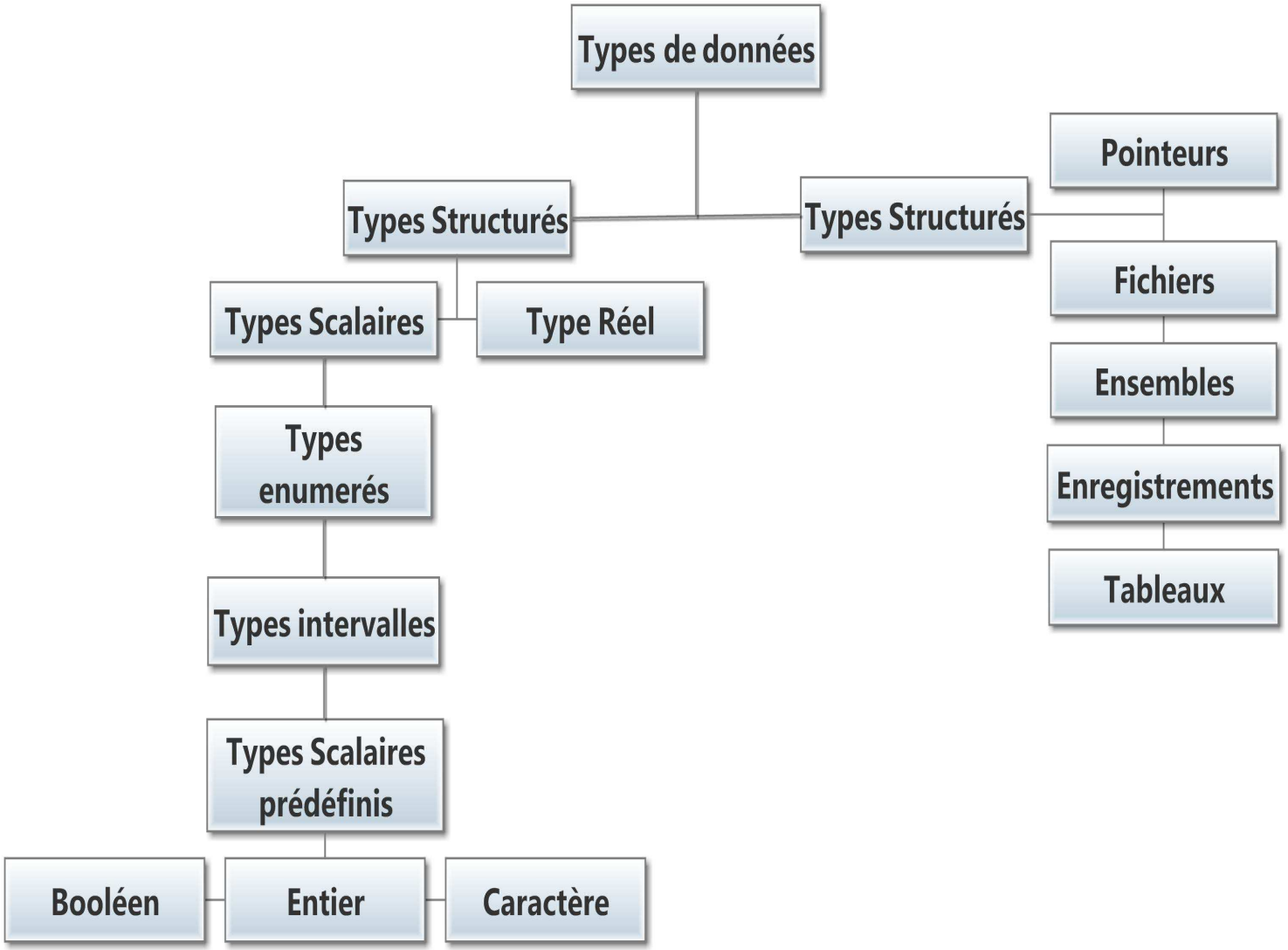
- Les Algorithmes sont élaborés à partir de composants divers permettant de spécifier en termes précis un schéma de résolution:
  - Caractères;
  - Libellé ou texte;
  - Opérateurs;
  - Mots clés;
  - Identificateurs;
  - Expressions;
  - Sous-Programmes ou sous-algorithme
  - etc...
- On distingue de manière générale, deux grands groupes de composants algorithmiques:
  - Les objets sur lesquels l'algorithme opère: les données (d'entrée et sortie), les valeurs numériques, caractères, chaîne de caractères, etc..;
  - Les éléments algorithmiques permettant la construction de l'algorithme: mots-clés, identificateurs, expressions, opérateurs, sous-programmes etc...

Les Objets Élémentaires: Constantes et Variables

Objets

- Ils représentent:
  - Les données ou informations fournies à l'algorithme;
  - Les résultats produits par l'algorithme;
  - Les données ou résultats intermédiaires servant aux manipulation internes (compteur, stockage des résultats intermédiaires, etc);
- On distingue de manière générale, deux catégories d'objets:
  - Les **Constantes**: dont les valeurs ne peuvent varier au cours de l'exécution de l'algorithme;
  - Les **Variables**: dont les valeurs peuvent varier lors de l'exécution de l'algorithme.
- Tout objet élémentaire est caractérisé par trois attributs:
  - un (**Identificateur**) qui est un nom symbolique utilisé pour manipuler l'objet;
  - un **Type** qui caractérise les valeurs que peut prendre l'objet ainsi que les actions autorisées sur celui-ci;
  - une **valeur**;
- En générale, l'Identicateur et le Type d'un objet sont fixes.

Classification de types de données



Les Eléments de base

- **opérateurs**
  - arithmétiques;
  - relationnels (comparaison);
  - logiques;
  - affectation;
  - fonctionnels
- **expressions;**
- **instructions**
  - simples;
  - structurées ou composées.
- **sous-programmes**
  - fonctions;
  - procédures.

Les Eléments de base: Opérateurs

Opérateurs arithmétiques

Nom	Symbole	Exemple
Addition	+	a+b
Soustraction	-	a-b
Opposé	-	-a
Produit	*	a*b
Division	/	a/b

Opérateurs relationnels

Nom	Symbole	Exemple
Égale	==	a==b
Différent	$\neq$	a $\neq$ b
Inférieur	<	a<b
Supérieur	>	a>b
Inférieur ou égal	$\leq$	a$\leq$b
Supérieur ou égal	$\geq$	a$\geq$b

Les Eléments de base: Opérateurs (suite)

Opérateurs Logiques

Nom	Symbole	Exemple
Négation	**non**	**non** a
Et	**et**	a **et** b
Ou	**ou**	a **ou** b

Opérateur d'affectation

Nom	Symbole	Exemple
Affectation	$\leftarrow$	a $\leftarrow$ b

Opérateurs fonctionnels

- Fonctions mathématiques standards;
- Fonctions de conversion de type;
- etc...;

Les Expressions

Définition

Une ***expression algorithmique*** est une combinaison des objets de l'algorithme et/ou des valeurs constantes (opérandes) à travers des opérateurs et/ou des sous-programmes.

Exemple:

Nom_Variable; 2*A+B/C; **sin**(A+5);
cos(Pi+0.3); (B-$\sqrt{4 * C}$/4);
(A=0 **et** B=1) **ou** (C>1 **et** **non** (D $\geq$ 0));

Priorité des opérations

- Lorsqu'on associe plusieurs opérations, celles-ci s'exécutent suivant un ordre de priorité bien défini. Suivant la priorité décroissante, on a l'ordre suivant:

Ordre	Opérateurs
0	( )
1	Appel de fonctions et procédures
2	Elevation à la puissance
3	*, /
4	+, -
5	==, $\neq$, <, >, $\geq$, $\leq$
6	**et**, **ou**, **non**

- En cas de priorité de même niveau, les opérations s'effectuent de la gauche vers la droite.

Les Instructions d'un algorithme

Définition:

Une **instruction** d'un algorithme est une directive précisant ce qu'il faire à un point précis de celui-ci.
Les instructions d'un algorithme décrivent plus ou moins en détail les calculs et/ou les traitements sur les données. Elles s'exécutent dans l'ordre séquentiel dans lequel elles apparaissent.

On distingue en général:

- Les instructions de déclaration;
- Les instructions de base (ou de traitement des données)
- Les instructions de structuration des algorithmes (ou instructions structurées ou composées).

Les Instructions de déclaration

Elles se trouvent dans la partie déclarative et permettent de:

- Déclarer les **Constantes**;
- Définir les **Types auxiliaires**;
- Déclarer les **Variables**;

Déclaration des constantes

Syntaxe:

Constante:

Nom_Const1 = Val1;
Nom_Const2 = Val2;
.....;
Nom_ConstN = ValN;

où Val1, Val2,...,ValN sont des valeurs constantes et/ou des résultats issus de l'évaluation des fonctions prédéfinies.

Exemple:

Constante:

pi = 3.14;
Nbre_Or = 1.618033;
Nmax = 100;

Définition de Types auxiliaires

Type auxiliaire:
Nom_type1 = description du Type1;
.....;
Nom_typeN = description du TypeN;

Exemple:

Type auxiliaire:
vecteur = **tableau** [1..Taille_Vect] **de réel**;
matrice = **tableau** [1..Nbre_ligne,1..Nbre_col] **de réel**;
point = **enregistrement de**:
x : **réel**;
y : **réel**;
fin;
Suite = **enregistrement de**:
Taille : **entier**;
Val : vecteur;
fin;

Déclaration des Variables

Syntaxe:

Variable:

Nom_var11, Nom_var12, ..., Nom_var1k : Type1;

.....;

Nom_varN1, Nom_varN2, ..., Nom_varNm : TypeN;

Exemple:

Variable:

A, B, C, Delta : **réel**;

i, j, k, n : **entier**;

u, v, w : vecteur;

M1, M2, M3 : matrice;

sexe : **caractère**;

Trouver : **booléen**;

S1,S2,S3,S4 : Suite;

Instructions de base

Elles sont constituées des:

- Instructions d'affectation;
- Instructions de lecture et d'écriture;
- appels de sous-programmes

Instructions d'affectation:

Les instructions d'affectation permettent d'attribuer à une variable, une valeur issue de l'évaluation d'une expression. Cette valeur doit être compatible avec le type de la variable.

Syntaxe:

Nom_Variable ← Expression;

Exemple:

A ← 5;
B ← 4*A + 3.14;
Lettre ← "A";
Test ← A==B;

Instructions de lecture et écriture:

Les instructions de lecture et d'écriture permettent d'établir la communication entre un programme et sont utilisateur: Acquérir les données nécessaires, renvoyer les résultats obtenus, etc...

Syntaxe:

Lire (Suite noms de variables séparées par des virgules);
Ecrire (Liste d'objets);

où Liste d'objets est une suite d'objets (constantes, variables, expression ou chaîne de caractères) séparés par des virgules.

Exemple:

Lire (A, B, C);
Delta ← B*B - 4*A*C;
Ecrire ("Le discriminant est: ", Delta);

Instructions de structuration ou structurées

Elle sont constituées des:

- Blocks d'instructions;
- Instructions répétitives (boucles "pour", "Tant Que" et "Répéter .... Jusqu'à");
- Les instructions conditionnelles (alternative simple, alternative complète, choix multiple).

...

Block d'instructions

C'est une séquence (suite d'instructions à exécuter dans un ordre donné) marquée par les délimiteurs **début** et **fin**.

Syntaxe:

début:
    instruction1;
    instruction2;
        .....;
    instructionN;
fin:

Exemple:

début
    Lire (A, B, C);
    Delta ← B*B - 4*A*C;
    Lettre ← "A";
    Ecrire ("Le discriminant est: ", Delta);
fin

Instructions répétitives: Boucle "Pour"

Syntaxe:

Pour Index = Ind_Déb (Pas) Ind_Fin **Faire**
début
instruction1;
instruction2;
.....;
instructionN;
fin

Exemple:

Données: N
S ← 0;
Pour i = 1 (1) N **Faire**
début
S ← S + 1/(i*i);
fin

Q: Que fait ce morceau d'algorithme?

Instructions répétitives: Boucle "Tant Que"

Syntaxe:

Tant que «Condition» **Faire**
début
instruction (s);
fin

Exemple:

Données: Theta
S ← 0;
i ← 0;
Tant que i < 1000 **Faire**
début
S ← S + **cos**(i*Theta);
i ← i+1;
fin

Q: Que fait ce morceau d'algorithme?

Instructions répétitives: Boucle "Répéter ... Jusqu'à"

Syntaxe:

Répéter
    instruction (s);
Jusqu'à «Condition»

Exemple:

Données: Theta
S ← 0;
i ← 0;
Répéter
    S ← S + **sin**(i*Theta);
    i ← i+1;
Jusqu'à i>1000

Q: Que fait ce morceau d'algorithme?

Instructions conditionnelles: Alternative Simple

Syntaxe:

Si «Condition» **Alors**
début
instruction (s);
fin

Exemple:

Données: a
S ← a;
Si a < 0 **Alors**
début
S ← -a;
fin

Q: Que fait ce morceau d'algorithme?

Instructions conditionnelles: Alternative Complète

Syntaxe:

Si «Condition» **Alors**
début
instruction (s);
fin
Sinon
début
instruction (s);
fin

Exemple:

Données: x
Si $x < 0$ Alors
début
$Y \leftarrow 0$;
fin
Sinon
début
$Y \leftarrow 1$;
fin

Q: Que fait ce morceau d'algorithme?

Instructions conditionnelles: Choix Multiple

Syntaxe:

```
Cas V =
    Val1,
      début
          instruction (s);
    Val2,
    ..............
    ValN,
      début
          instruction (s);
      fin
    Sinon
      début
          instruction (s);
      fin
fin
```

Structure générale d'un programme en LEA

```
Programme Nom_du_Prog
    (***  Déclarations  ***)
      (*  Objectif: ..........*)
      (*              ..........*)
      (*  Donnée(s): ..........  *)
      (*                ..........*)
      (*  Résultat(s): ..........*)
      (*              ..........*)
      Constante : ................;
                  ................;
      Type Auxiliare : ................;
                       ................;
      Variable : ................;
                 ................;
     (*** Algorithme ou Corps du Prog ***)
    Début  (* de Nom_du_Programme *)
           Instruction(s) 1;
           Instruction(s) 2;
           Instruction(s) 3;
           ........................;
           Instruction(s) N;
    Fin  (* de Nom_du_Programme *)
    (*** Sous-Programmes ***)
    ..........................................
```

Manipulation des fichier en LEA

- Un fichier est un objet de l'algorithme qui doit être déclaré et activé avant toute utilisation.
- Déclaration des fichier:
  Nom_Fichier : **fichier_texte**;
- Activation du fichier: Elle consiste à faire correspondre à ce dernier un fichier physique par l'action d'attachement où d'assignation comme suit:
  Attacher(Nom_Fichier, "Chemin d'accès au fichier");
  Exemple: **Attacher**(Fic1, "C:\Perso\TP\Mydata.txt");
- Ouverture en lecture et écriture du fichier:
  Ouvrirlecture(Nom_Fichier); **Lire**(Nom_Fichier, liste des variables);
  Ouvrirecriture(Nom_Fichier); **Ecrire**(Nom_Fichier, liste d'objets);
- Contrôle de fin de fichier: **EOF**(Nom_Fichier);
- Fermeture du Fichier: **Fermer**(Nom_Fichier);

Sous Programmes en LEA

- Un fichier est un objet de l'algorithme qui doit être déclaré et activé avant toute utilisation.
- Déclaration des fichier:
  Nom_Fichier : **fichier_texte**;
- Activation du fichier: Elle consiste à faire correspondre à ce dernier un fichier physique par l'action d'attachement où d'assignation comme suit:
  Attacher(Nom_Fichier, "Chemin d'accès au fichier");
  Exemple: **Attacher**(Fic1, "C:\Perso\TP\Mydata.txt");
- Ouverture en lecture et écriture du fichier:
  Ouvrirlecture(Nom_Fichier); **Lire**(Nom_Fichier, liste des variables);
  Ouvrirecriture(Nom_Fichier); **Ecrire**(Nom_Fichier, liste d'objets);
- Contrôle de fin de fichier: **EOF**(Nom_Fichier);
- Fermeture du Fichier: **Fermer**(Nom_Fichier);

Définition: Algorithmes Récursifs

Un sous programme est dit récursif s'il s'appelle lui même, de façon directe (appel direct) ou indirecte (appel à un autre sous programme qui fait de nouveau appel à ce dernier).

- Les sous programmes récursifs s'appliquent généralement aux problèmes qui sont de nature récursive. Comme:
  - Les formules mathématiques récurrentes;
  - Tri des données dans les structures (tableaux, listes, arbres, graphes)
  - etc
- L'exécution des sous programmes récursifs se fait généralement en deux phases en utilisant une structure de pile qui sert à sauvegarder les variables intermédiaires aux moment ou le sous programme s'appelle lui même:
  - **L'empilement des tâches**;
  - **Le désempilement des tâches**.
- La profondeur de la pile étant finie, un bon sous-programme récursif doit avoir une condition d'arrêt de la récursion.
- La pile est du type **FILO**

Avantages et Inconvénients

- **Avantages**:
  - Codification compacte et réduite;
  - Permet de résoudre un problème en résolvant une ou des pièces réduites du même problème en utilisant le même code.
  - etc
- **Inconvénients:**
  - Surveiller la profondeur disponible de la pile (le système pouvant éjecter le processus par dépassement des ressources autorisées);
  - Le temps d'exécution est plus long, à cause des sauvegardes et de récupérations de tâches sur la pile.

Exemple: Calcul de la factorielle d'un entier

Type de récursivité

- Récursivité simple ou linéaire.
- Récursivité multiple.
- Récursivité mutuelle, indirecte ou croisée.

Type de récursivité (suite)

Récursivité simple ou linéaire.

Un algorithme récursif est dit simple ou linéaire, si chaque cas qu'il distingue se résout en au plus un appel récursif.

Exemple.

- Calcul de la factorielle d'un entier;
- Calcul du PGCD de deux entiers positifs.

Type de récursivité (suite)

Récursivité multiple.

Un algorithme récursif est dit multiple, si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Exemple:

- Suite de Fibonacci.

$F_0 = 0;\ F_1 = 1;\ F_{n+2} = F_{n+1} + F_n,\ \forall n \in \mathbb{N}.$

- Fonction d'Ackermann-Péter

$$A(m,n) = \begin{cases} n+1 \ \text{ si } m = 0 \\ A(m-1,1) \ \text{ si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) \ \text{ si } m > 0 \text{ et } n > 0 \end{cases}$$

1) Calculer $A(0,4),\ A(1,2),\ A(2,1)$ et $A(3,0)$.

2) Ecrire une fonction permettant de calculer $A(m,n)$ pour m et n positifs.

Type de récursivité (suite)

$$A(0,4) = 5$$

$$A(1,2) = 4$$

$$A(2,1) = 5$$

Pour le calcul de $A(4,1)$ le nombre d'appels récursifs est supérieur à **500!!!**

Type de récursivité (suite)

Récursivité mutuelle, indirecte ou croisée.

Deux algorithmes sont dits mutuellement récursifs si l'un fait appel à l'autre et vis versa.

Exemples:

- Parité d'un entier:

$P(n)$ = prédicat de test de parité sur l'entier n;
$I(n)$ = prédicat de test d'«imparité» sur l'entier n.

```
P(n)
Si n=0 alors
  P(n) = VRAI
sinon
  P(n) = I(n-1)
Fin
```

```
I(n)
Si n=0 alors
  I(n) = FAUX
sinon
  I(n) = P(n-1)
Fin
```

Outline

Complexité des Algorithmes

- **Complexité d'un algorithme** : Coût de ce dernier. Elle est mesurée en fonction d'un paramètre entier n caractérisant le problème à résoudre.
- **But**:
  - Evaluer l'efficacité des algorithmes;
  - Comparer les performances des algorithmes effectuant les mêmes tâches.
- **Propriétés**:
  - Indépendance vis-à-vis de l'ordinateur utilisé;
  - Indépendance vis-à-vis du langage de programmation utilisé pour sa traduction;
  - Indépendance vis-à-vis du programmeur;
  - Indépendance vis-à-vis des détails d'implementation;
  - etc...

Complexité des Algorithmes (Suite)

- **Paramètres essentielles:**
  - Espace mémoire requis (de moins en moins considérer actuellement, mais pris en compte en cas de récursivité);
  - Temps d'exécution (Décomptes du nombre d'opérations élémentaires pertinentes pour la résolution du problème considéré);

  Ainsi, on a:
- **Complexité en temps**: C'est une fonction f de n qui permet de mesurer le temps de calcul pour la mise en oeuvre de l'algorithme;
- **Complexité en mémoire**: C'est une fonction f de n qui permet de mesurer la quantité de place mémoire nécessaire à la mise en oeuvre de l'algorithme.

Complexité des Algorithmes (Suite)

- **Ordre de grandeur**:
  Ne pouvant pas toujours déterminer avec exactitude le nombre d'opérations élémentaires, on donne souvent l'ordre de grandeur de celui-ci en fonction du nombre de paramètres caractéristiques n du problème.
- **Ordre de grandeur des fonctions**: Notations O, Θ et Ω.
  Soit f une fonction définie sur $\mathbb{N}$

$$O(f) = \{g : \exists c > 0,\ n_0 \in \mathbb{N} \text{ tel que } \forall n \geq n_0,\ g(n) \leq c.f(n)\};$$

$$\Omega(f) = \{g : \exists c > 0,\ n_0 \in \mathbb{N} \text{ tel que } \forall n \geq n_0,\ g(n) \geq c.f(n)\};$$

$$\begin{aligned}\Theta(f) \quad = \quad & \{g : \exists c_1 > 0, \exists c_2 > 0,\ n_0 \in \mathbb{N} \text{ tel que} \\ & \forall n \geq n_0,\ c_1.g(n) \leq f(n) \leq c_2.g(n)\};\end{aligned}$$

Complexité des Algorithme (Suite)

- **Borne supérieure asymptotique:** $f(n)$ est une borne asymptotique supérieure de $g(n)$ (ou $g(n)$ est de borne asymptotique supérieure $f(n)$) si

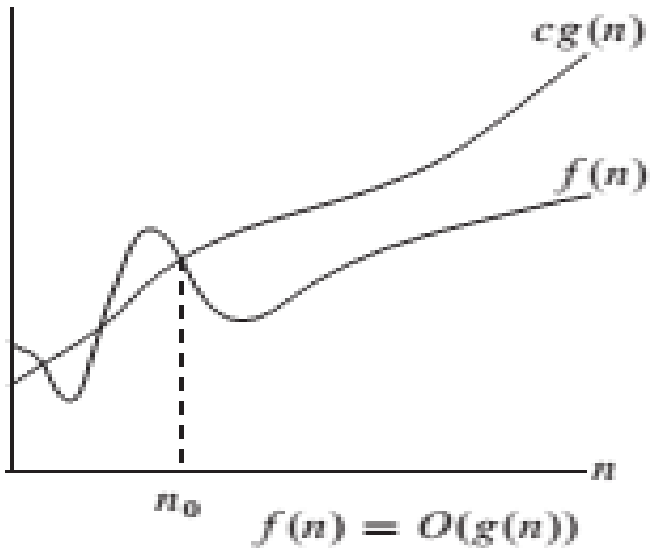
$$g(n) = O(f(n))$$

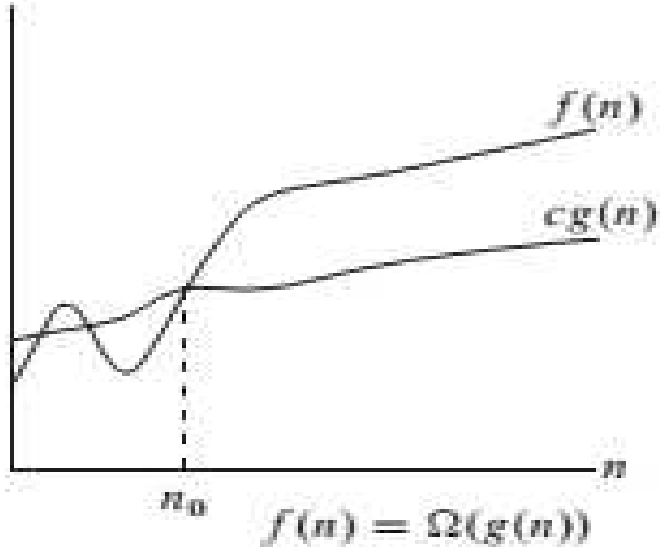
- **Borne inférieure asymptotique:** $f(n)$ est une borne asymptotique inférieure de $g(n)$ (ou $g(n)$ est de borne asymptotique inférieure $f(n)$)si

$$g(n) = \Omega(f(n))$$

- **Borne asymptotique:** $f(n)$ est une borne asymptotique $g(n)$ si

$$g(n) = \Theta(f(n))$$





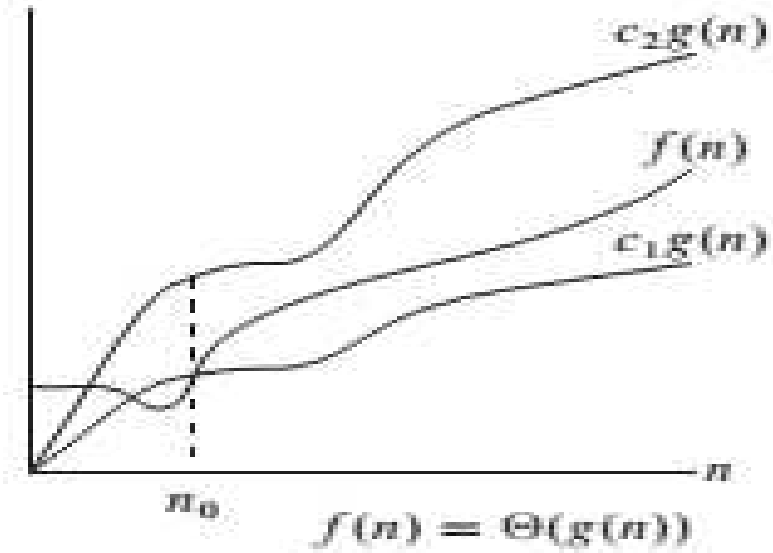


Illustration des notations asymptotiques

Quelques Propriétés des notations asymptotiques

- **Réflexivité:**
  - $f = O(f)$;
  - $f = \Omega(f)$;
  - $f = \Theta(f)$.
- **Symétrie:**
  - $f = \Theta(g) \quad ssi \quad g = \Theta(f)$
- **Transitivité:**
  - $Si\ f = O(g)\ et\ g = O(h)\ \ alors\ \ f = O(h)$;
  - $Si\ f = \Omega(g)\ et\ g = \Omega(h)\ \ alors\ \ f = \Omega(h)$;
  - $Si\ f = \Theta(g)\ et\ g = \Theta(h)\ \ alors\ \ f = \Theta(h)$.
- **Produit par un scalaire:**
  - $Soit\ \lambda > 0.\ \ \lambda O(f) = O(\lambda f) = O(f)$
- **Somme et produit de fonctions:**
  - $O(f) + O(g) = O(\max(f, g))$;
  - $O(f).O(g) = O(f.g)$;

Complexité des Algorithmes (Suite)

- Dans l'étude de la complexité des algorithmes, on distingue de manière générale, les cas suivants:
  - Cas le plus favorable;
  - Cas le plus défavorable;
  - Cas moyen (parfois difficile à déterminer).
- **Le temps d'exécution d'un algorithme est un $\Theta(f(n))$ si et seulement si son temps d'exécution dans le cas le plus défavorables est un $O(f(n))$ et dans le cas le plus favorable est un $\Omega(f(n))$.**

Complexité des Algorithmes (Suite)

Classes de complexité généralement rencontrés

	Notation	Type de complexité	pour $n = 100$ à 10^8 ops/s
1	$O(1)$	Constante	Temps constant
2	$O(\log(n))$	Logarithmique	10^{-7} seconde
3	$O(n)$	Linéaire	10^{-6} seconde
4	$O(n\log(n))$	Quasilinéaire	10^{-5} seconde
5	$O(n^2)$	Quadratique	10^{-4} seconde
6	$O(n^3)$	Cubique	10^{-2} seconde
7	$O(n^p)$; $p > 1$	Polynomiale	11 jours si $p = 7$
8	$O(a^n)$; $a > 1$	Exponentielle	10^{14} années si $a = 2$
9	$O(n!)$	Factorielle	10^{142} années

Rappels sur les sommations.

- Les sommations interviennent naturellement dans l'analyse de la complexité des algorithmes itératifs ainsi que dans la résolution de certaines récurrences.
- Comme mentionné plus haut on pourra se contenter dans certains cas de l'ordre de grandeur des sommations.
- **Série constante:** Soient m et n deux entiers. On a

$$\sum_{i=m}^{n} 1 = \max(n - m + 1, 0).$$

- Soit $n \in \mathbf{IN}$,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}; \qquad \sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}$$

- Soit $x \neq 1$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}; \qquad \sum_{i=0}^{n} ix^i = \frac{(n-1)x^{n+1} - nx^n + x}{(x-1)^2}$$

- **Approximation par une intégrale**: Soit f une fonction strictement croissante, on a:

$$\int_0^n f(x)dx \leq \sum_{i=1}^{n} f(i) \leq \int_1^{n+1} f(x)dx.$$

- Soit m et n deux entiers tels que $1 \leq m \leq n$,

$$\sum_{i=m}^{n} f(i) = \sum_{i=0}^{n} f(i) - \sum_{i=0}^{m-1} f(i)$$

- **Série harmonique**: Soit n un entier strictement positif, on a:

$$\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$$

- **Coût d'une conditionnelle**: On considère la structure

  Si (Cond) **Alors**

  ACTION 1;

  Sinon

  ACTION 2;

  FinSi

  Si $O(h(n))$, $O(f(n))$ et $O(g(n))$ désignent respectivement les complexités de l'évaluation de $Cond$, *ACTION 1* et *ACTION 2*, alors la complexité de la structure conditionnelle ci-dessus est

$$\max\Big(O(h(n)), O(f(n)), O(g(n))\Big).$$

- **Coût d'une boucle Pour**: Dans la boucle

  Pour i=Ideb (pas) Ifin **faire**

  début

  ACTION(i);

  fin

  Si $c(i)$ désigne le coût de l'exécution de ACTION(i) et $k = E\left((Ifin - Ideb)/pas\right) + 1$ désigne le nombre de passage dans la boucle. Soit $U = \{u_i\}_{i=1}^{k}$ la suite finie définie par $u_1 = Ideb$, $u_{i+1} = U_i + pas$ $(1 \leq i \leq k-1)$ alors le coût de la boucle est:

$$C = \sum_{j \in U} c(j)$$

- Si $c(i) = c$, une constante, alors $C = c.k$.
- Si $pas = 1$ alors

$$C = \sum_{Ideb}^{Ifin} c(i)$$

- **Coût d'une boucle Tant que**: Dans la boucle

Tant que $C(x)$ **faire**
début
ACTION(x);
fin

Soit x_0 la valeur initiale de la donnée x et $x_1, x_2, ..., x_k$ les valeurs qu'elle prend successivement à chaque étape, x_{k+1} la première valeur pour laquelle la condition $C(x_{k+1})$ n'est plus satisfaite;
Soient $c(i)$ le coût de la condition $C(x_i)$ et $f(i)$ le coût de l'exécution de ACTION(x_i). Alors le coût de la boucle précédente est:

$$C = \sum_{i=0}^{k+1} c(i) + \sum_{i=0}^{k} f(i)$$

.

Cas des Algorithmes récursifs:

La complexité d'un algorithme récursif s'obtient par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

- **Exemple 1**: Calcul de la factorielle

```
fact(n):
    Début
        Si n==0 alors
            Renvoyer(1);
        sinon
            Renvoyer(n*fact(n-1));
        fsi
    Fin
```

Le coût $C(n)$ en termes de multiplications est:

$$\begin{cases} C(0) = 0 \\ C(n) = C(n-1) + 1, n \geq 1 \end{cases}.$$

Soit: $C(n) = n$

- **Exemple 2**: Problème de la Tour d'Hanoï

```
HANOI(n,D,A,I):
    Début
        Si n==1 alors
            Déplacer de D vers A;
        sinon
            HANOI(n-1,D,I,A);
            Déplacer de D vers A;
            HANOI(n-1,I,A,D);
        fsi
    Fin
```

Le coût $C(n)$ en termes du nombre de déplacement dépend de n et est donné par:

$$C(1) = 1 \text{ et } C(n) = 1 + 2C(n-1),\ n \geq 2$$

Soit: $C(n) = 2^n - 1$

- Le coût d'un algorithme s'exprime généralement sous la forme d'une équation récurrente.

Cas des Algorithmes de Recherche et de Tri

Performance des algorithmes de recherche et de tri: On considère les algorithmes de recherche et de tri appliqués au tableaux de taille N. Les opérations considérées sont:

- Les comparaisons sur les éléments du tableau. On note $C(N)$ le nombre de comparaisons.
- Les déplacements des éléments du tableau. On note $D(N)$ le nombre de déplacements.

Recherche séquentielle dans un tableau non trié On recherche la position d'un réel x dans le tableau non trié T de taille N.

Fonction RechercheNonTrie (T : tableau, Nb_elts :**entier**,
x : **réel**) : **entier**

(* **Objectif**: Recherche la position de l'élément x dans le tableau non trié de réels T*)
(* **Paramètres entrants**: T: tableau de réels; Nb_elts: Nbre d'éléments du tableau T; x: réel dont on cherche la position dans T *)
(* **Résultat:** position de x si x est dans T, -1 sinon *)

Variable:

i: **entier**;

Début

$i \leftarrow 1$;

Tant que $(i \leq N)$ **et** $(T[i] \neq x)$ **faire**

$i \leftarrow i + 1$;

fTantque

Si $i == N + 1$ **alors**

RechercheNonTrie $\leftarrow -1$;

Sinon

RechercheNonTrie $\leftarrow i$

fSi

Fin

- **Cas favorable**: $x = T[1]$; $C(N) = O(1)$;
- **Cas défavorable**: $x \notin T$; $C(N) = O(N)$.
- **Cas moyen**: x se trouve au milieu du tableau T; $C(N) = O(N/2)$

Recherche séquentielle dans un tableau trié

On recherche la position d'un réel x dans le tableau trié T de taille N.

Fonction RechercheLinTrie (T : tableau, Nb_elts :**entier**,
x : **réel**) : **entier**
(* **Objectif**: Recherche la position de l'élément x dans le tableau trié de réels T*)
(* **Paramètres entrants**: T: tableau de réels; Nb_elts: Nbre d'éléments du tableau T; x: réel dont on cherche la position dans T *)
(* **Résultat:** position de x si x est dans T, -1 sinon *)

Variable:
i: **entier**;

Début

Si $x < T[1]$ **ou** $x > T[N]$ **alors**
RechercheLinTrie $\leftarrow -1$
fSi
$i \leftarrow 1$;
Tant que $(x > T[i])$ **faire**
$i \leftarrow i + 1$;
fTantque
Si $x \neq T[i]$ **alors**
RechercheLinTrie $\leftarrow -1$;
Sinon
RechercheLinTrie $\leftarrow i$
fSi

Fin

- **Cas favorable**: $x > T[N]$ ou $x \leq T[1]$; $C(N) = O(1)$;
- **Cas défavorable**: $x \notin T$ et $T[N-1] < x < T[N]$; $C(N) = O(N)$.
- **Cas moyen**: x se trouve au milieu du tableau T; $C(N) = O(N/2)$

Recherche dichotomique dans un tableau trié de taille N

On recherche la position d'un réel x dans le tableau trié T de taille N.

Fonction RechercheBinTrie (T : tableau, Nb_elts :**entier**,
x : **réel**) : **entier**
(* **Objectif**: Recherche la position de l'élément x dans le tableau trié de réels T par une approche dichotomique *)
(* **Paramètres entrants**: T: tableau de réels; Nb_elts: Nbre d'éléments du tableau T; x: réel dont on cherche la position dans T *)
(* **Résultat:** position de x si x est dans T, -1 sinon *)

Variable:
$indG$, $indD$, $mileu$: **entier**;

Début

$indG \leftarrow 1$; $IndD \leftarrow N$;
Tant que $(indG \leq indD)$ **faire**
début
$milieu \leftarrow div(indG + indD, 2)$;
Si $(x == T[milieu])$ **alors**
Renvoyer(milieu);
fSi
Si $(x < T[milieu])$ **alors**
$IndD \leftarrow milieu - 1$;
Sinon
$IndG \leftarrow milieu + 1$;
fSi
fin
Renvoyer(-1);

Fin

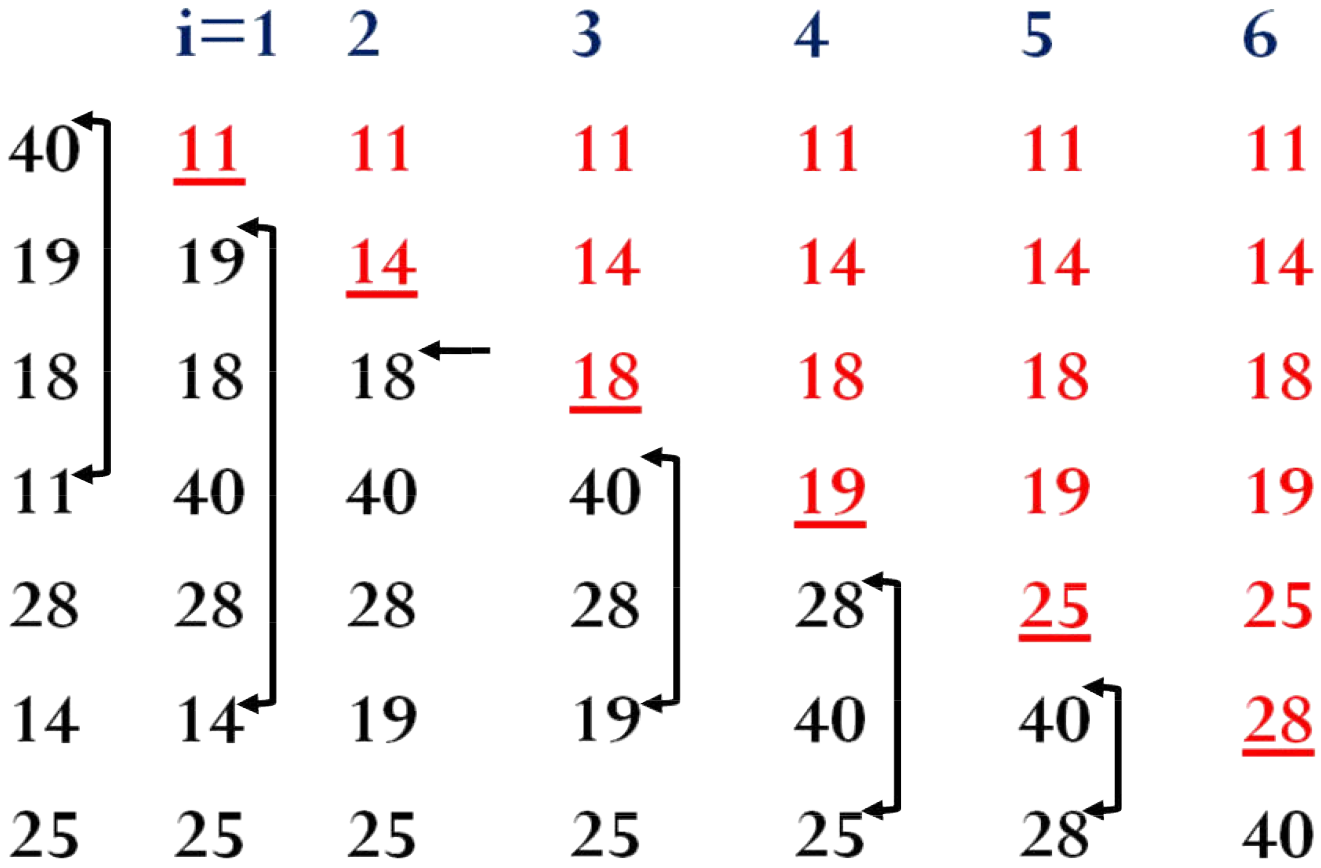
- **Cas favorable**: x est au milieu de T; $C(N) = O(1)$;
- **Cas défavorable**: $x \notin T$; $C(N) = O(log_2(N))$.

Algorithmes de Tri

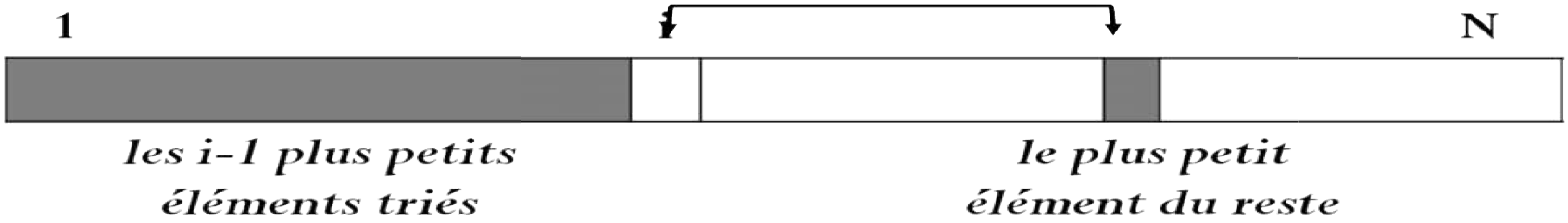
- Le tri consiste à réarranger une permutation de n objets suivant un ordre croissant ou décroissant.
- Plusieurs algorithmes de tri:
  - Tri par sélection;
  - Tri par insertion;
  - Tri bulle;
  - Tri fusion;
  - Tri par segmentation (Quick Sort),...

Tri sélection d'un tableau de taille N

Principe: Chercher le plus petit (le plus grand) élément et le mettre au début (à la fin). Répéter le processus sur le sous vecteur non trié



A l'étape i, on a la configuration suivante:



Une procédure commune aux algorithmes de tri: l'échange de valeurs

Procédure Echange(Var x,y:réel);
(* **Objectif**: échanger les valeurs de 2 variables réelles x et y;
Paramètres rentrants: x, y initiaux;
Paramètres sortants: x, y permutés;*)
Variable:
aux: **réel**;
Début (* de Echange*)
$aux \leftarrow x$;
$x \leftarrow y$;
$y \leftarrow aux$;
Fin (* de Echange*)

Procédure TriSelection (Var T : tableau, Nb_elts :entier);
(* **Objectif**: Tri par ordre croissant d'un vecteur de nombres réels par sélection;
Paramètres entrants: T: tableau non trié; Nb_elts: Nbre d'éléments du tableau T;
Paramètres sortants: T, tableau trié*)
Variable:
$i,\ j,\ pp$: **réel**;

Début

Pour $i = 1(1)N - 1$ **faire**

début

$pp \leftarrow i;$

Pour $j = i + 1(1)N$ **faire**

début

Si $(T[j] < T[pp])$ **alors**

$pp \leftarrow j;$

fSi

fin

Si $pp \neq i$ **alors**

$Echange(T[i], T[pp])$

fSi

fin

Fin

Tri sélection d'un tableau de taille N

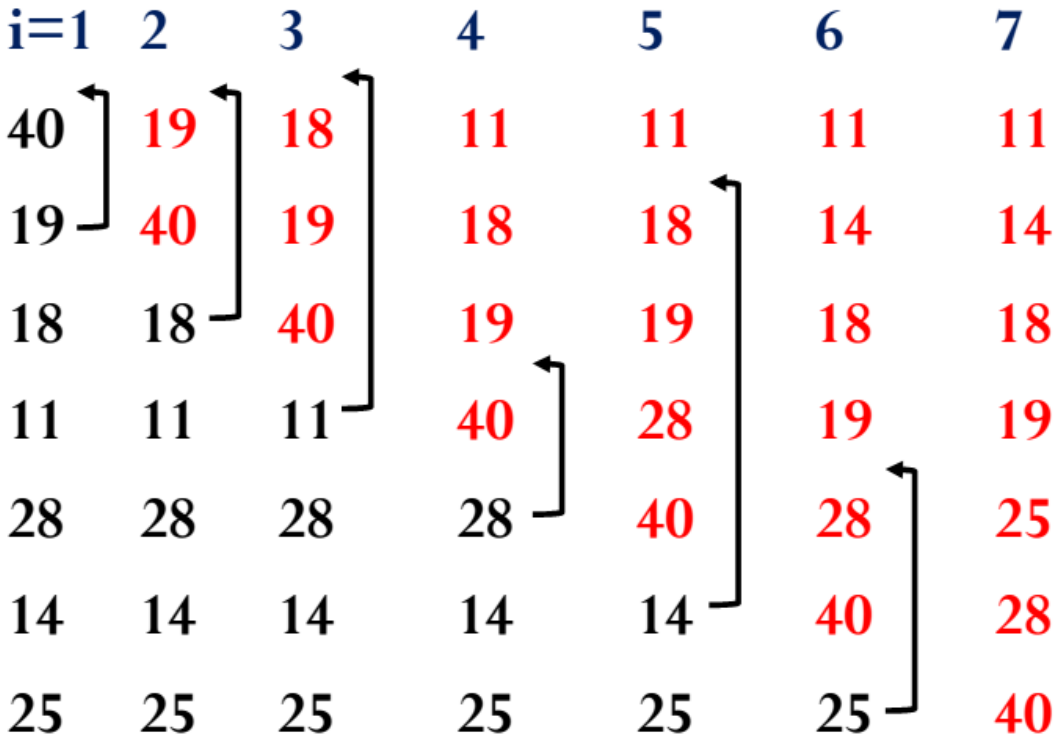
- A l'étape i de la méthode, on effectue $N - i$ comparaisons et 1 déplacement. D'où:

$$C(N) = \sum_{i=1}^{N-1} N - i = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = O(N^2)$$

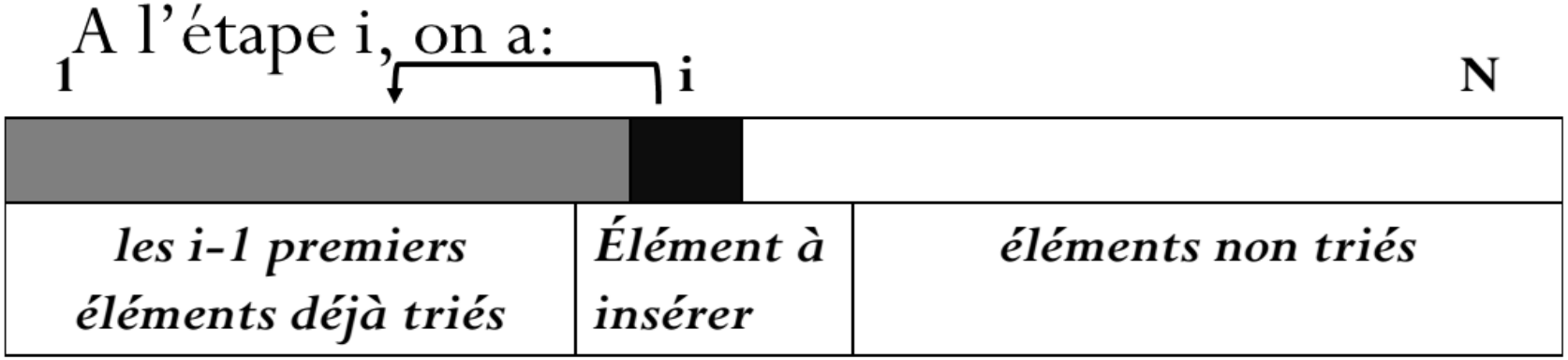
$$D(N) = \sum_{i=1}^{N-1} 1 = N - 1 = O(N)$$

Tri par insertion d'un tableau de taille N

Principe: On trie les deux premiers éléments dans le bon ordre, puis le 3è est inséré à sa place parmi les deux autres. A l'étape i, les i-1 premiers éléments sont déjà triés. L'élément i est alors inséré à sa place dans la partie trié du tableau.



Tri par insertion d'un tableau de taille N




```
Procédure TriSelection (Var T : tableau,
  Nb_elts :entier);
(* Objectif: Tri par ordre croissant d'un
  vecteur de nombres réels par l'algorithme
  du tri par insertion;
  Paramètres entrants:
  T: tableau non trié;
  Nb_elts: Nbre d'éléments du tableau T;
  Paramètres sortants: T, tableau trié*
```

Tri par insertion d'un tableau de taille N

Variable:

$i,\ j$: **entier**;
Val : **réel**;

Début (* TriInsertion *)

Pour $i = 2(1)N$ **faire**

début

$Val \leftarrow T[i]$;
$j \leftarrow i$;
Tant que $j > 1$ **et** $(T[j-1] > Val)$ **faire**
début
$T[j] \leftarrow T[j-1]$;
$j \leftarrow j-1$;
fin
$T[j] \leftarrow Val$;

fin

Fin (* TriInsertion *)

Tri par insertion d'un tableau de taille N

- **Cas favorable:** Le tableau est trié et à chaque étape i, on effectue 1 comparaison et 0 déplacement. D'où:

$$C(N) = N - 1 = O(N) \ \ et \ \ D(N) = 0$$

- **Cas défavorable:** Le tableau est trié dans l'ordre inverse. A chaque étape, l'insertion se fait au début de la partie triée: a l'étape i, on effectue $i-1$ comparaisons et $i-1$ déplacements. D'où:

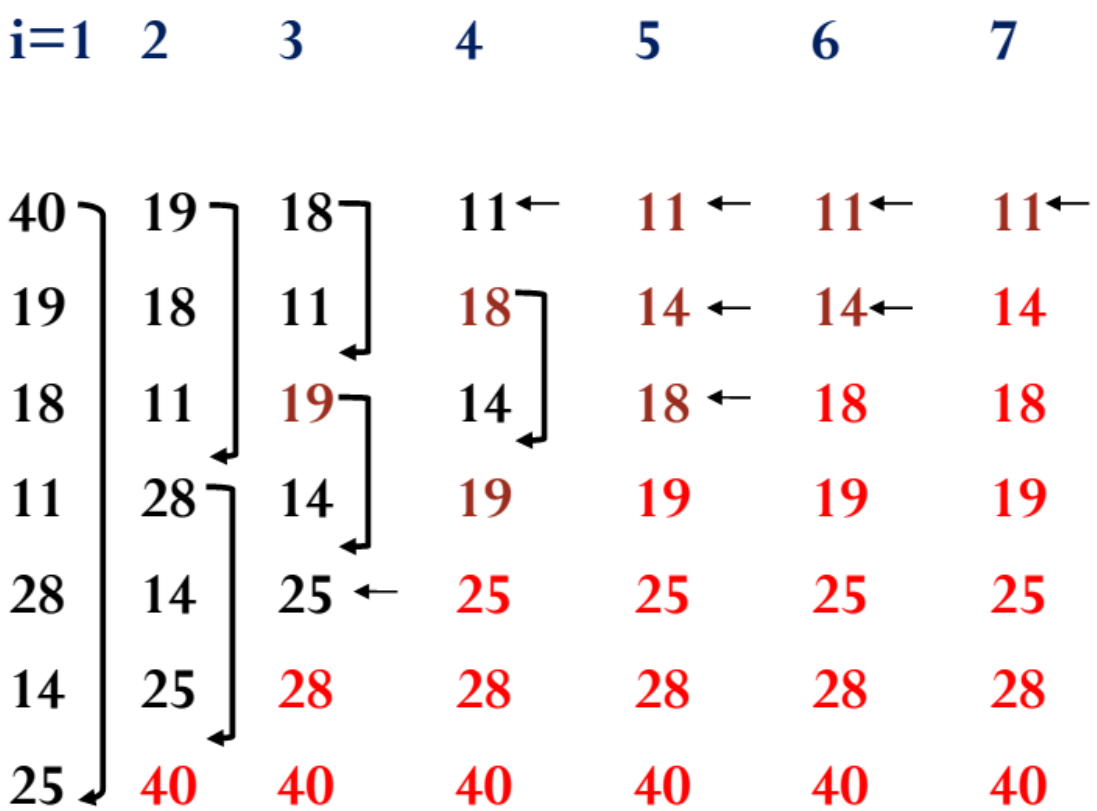
$$C(N) = D(N) = \sum_{i=2}^{N} i - 1 = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = O(N^2/2)$$

- **Cas moyen:** A chaque étape i, l'insertion se fait au milieu de la partie triée. On effectue alors $i/2$ comparaisons et $i/2$ déplacements. D'où:

$$C(N) = D(N) \approx \sum_{i=2}^{N} i/2 = \sum_{i=1}^{N-1} i/2 = \frac{N(N-1)}{4} = O(N^2/4)$$

Tri Bulle d'un tableau de taille N

Principe: A l'aide des comparaisons successives, on remonte le plus grand élément de la partie non triée du vecteur pour le positionner en début de la partie triée.



Tri Bulle d'un tableau de taille N

```
Procédure TriBulle (Var T : tableau,
  Nb_elts :entier);
(* Objectif: Tri par ordre croissant d'un
  vecteur de nombres réels par l'algorithme
  du tri à bulle;
  Paramètres entrants:
  T: tableau non trié;
  Nb_elts: Nbre d'éléments du tableau T;
  Paramètres sortants: T, tableau trié*)
Variable locale:
  i, j: entier;
  flag : booléen; (* Permet de
     stopper le processus lorsque
                le vecteur est déjà trié*)
```

Tri Bulle d'un tableau de taille N

Début

$i \leftarrow N-1$;
Flag $\leftarrow$ **Faux**;
Tant que (**non** Flag) **faire**
début
Flag $\leftarrow$ **Vrai**;
Pour $j = 1(1)i$ **faire**
début
Si $(T[j] > T[j+1])$ **alors**
Echange(T[j],T[j+1]);
Flag $\leftarrow$ **Faux**;
fSi
fin
$i \leftarrow i-1$;
fin

Fin

Tri Bulle d'un tableau de taille N

- **Cas favorable:** Le tableau est trié et à chaque étape i, on effectue 1 comparaison et 0 déplacement. D'où:

$$C(N) = N - 1 = O(N) \;\; et \;\; D(N) = 0$$

- **Cas défavorable:** Le tableau est trié dans l'ordre inverse. A chaque étape, le plus grand élément est au début de la partie non triée: a l'étape i, on effectue $N - i$ comparaisons et $N - i$ déplacements. D'où:

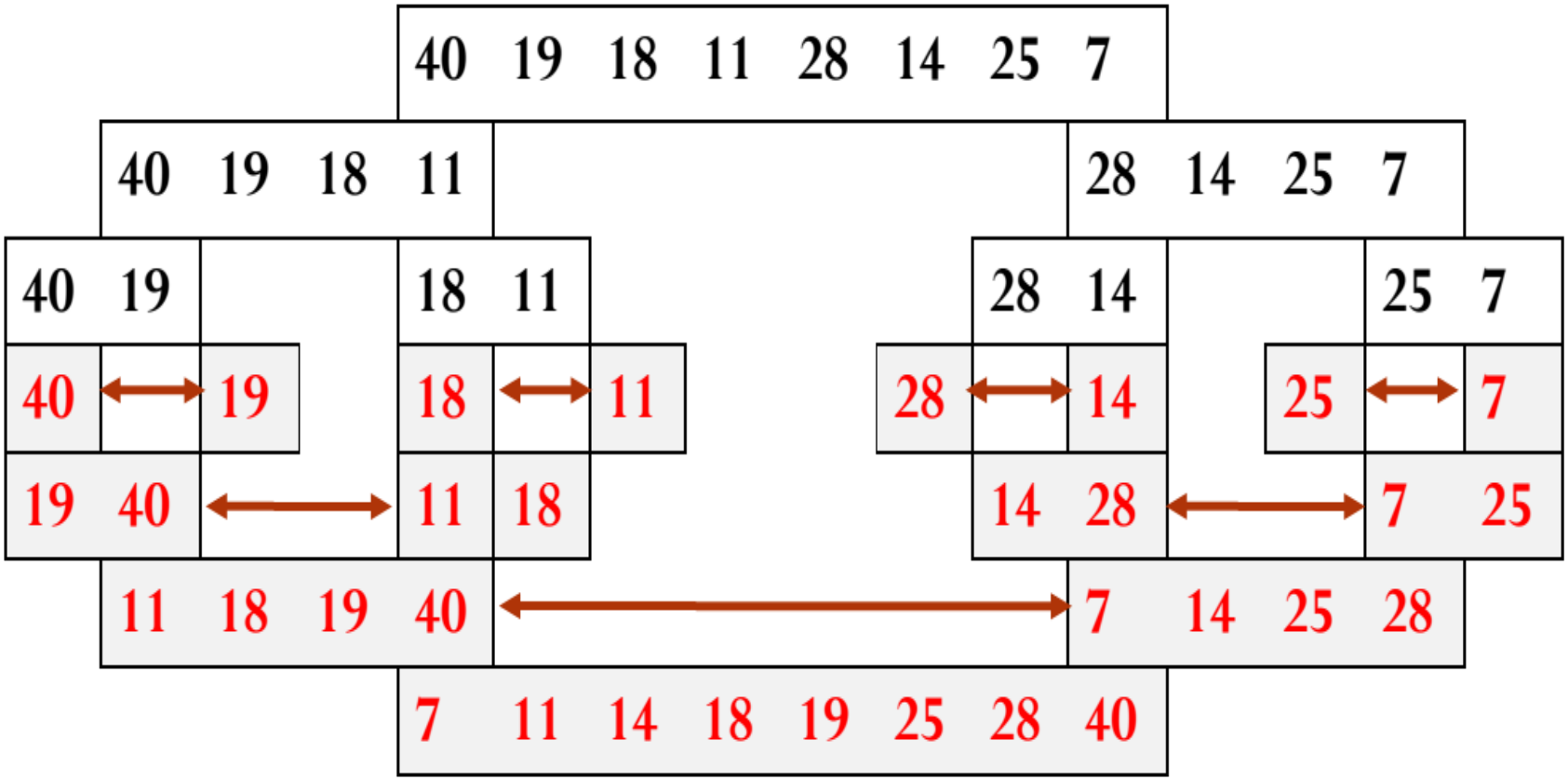
$$C(N) = D(N) = \sum_{i=2}^{N-1} N - i = \sum_{i=1}^{N-2} i = \frac{(N-1)(N-2)}{2} = O(N^2/2)$$

- **Cas moyen:** A chaque étape i, la valeur maximale se trouve à peu près au milieu de la partie non triée. On effectue alors $(N - i + 1)/2$ comparaisons et $(N - i + 1)/2$ déplacements. D'où:

$$C(N) = D(N) \approx \sum_{i=2}^{N-1} (N-i+1)/2 = \sum_{i=1}^{N-2} (i+1)/2 = \frac{N(N-1)}{4} = O(N^2/4)$$

Tri Fusion d'un tableau de taille N

Principe: Découper le tableau à trier en deux, trier chaque partie, puis fusionner en respectant l'ordre.



Tri Fusion d'un tableau de taille N

Une étape importante: la fusion de deux vecteurs triés en respectant l'ordre

```
Procédure Fusion (T1,T2 : tableau,
  Nb1,Nb2 :entier, Var T : tableau, Var
  Nb_elts:entier);
(* Objectif: Fusionner deux tableaux triés
 en respectant l'ordre;
 Paramètres entrants:
 T1, T2: tableaux triés à fusionner;
 Nb1, Nb2: Nbres d'éléments des tableaux
 T1 et T2 respectivement;
 Paramètres sortants: T, tableau trié issu
 de la fusion; Nb_elts: nbre d'éléments de
 T*)
```

Tri Fusion d'un tableau de taille N

```
Variable locale:
  i, j, k: entier;
Début (* Fusion *)
  Nb_elts ← Nb1 + Nb2;
   k ← 1; i ← 1; j ← 1;
  Tant que (i ≤ Nb1) et (j ≤ Nb2) faire
        début
              Si T1[i] ≤ T2[j] alors
                    début
                          T[k] ← T1[i];
                          i ← i+1;
                    fin
              Sinon
                    début
                          T[k] ← T2[j];
                          j ← j+1;
                    fin
               k ← k+1;
        fin
```

Tri Fusion d'un tableau de taille N

```
Tant que (i ≤ Nb1) faire
    début
        T[k] ← T1[i];
        i ← i+1;
        k ← k+1
    fin

Tant que (j ≤ Nb2) faire
    début
        T[k] ← T2[j];
        j ← j+1;
        k ← k+1
    fin
Fin (* Fusion *)
```

Tri Fusion d'un tableau de taille N

- **Tâche importante: fusionner les sous parties triées T[inf..milieu] et T[milieu+1..sup] pour obtenir la partie triée T[inf..sup] du vecteur T[1..Nb_elts].**
- **On fait une adaptation de l'algorithme précédent:**

```
Procedure Fusion1(Var T:tableau, inf, sup, milieu)
(*Objectif: fusionner les sous parties triées T[inf..milieu] et T[milieu+1..sup] pour obtenir la partie triée T[inf..sup] du vecteur T[1..Nb_elts].
Paramètres entrants:
  T: tableau;
  1<=inf<=milieu<sup<=Nb_elts: bornes des sous parteir à fusionner;
Paramètres sortants: T, tableau avec la partie T[inf...sup] triée*)
```

Tri Fusion d'un tableau de taille N

```
Variable locale:
  n1,n2,nb,i: entier;
  T1,T2,T3:tableau;
Début (* Fusion1 *)
  n1 ← milieu - inf +1;
  n2 ← sup - milieu;
  (* copie des parties à fusionner *)
  Pour i=1(1)n1 faire
        T1[i] ← T[i+inf-1];
  Pour i=1(1)n2 faire
        T2[i] ← T[i+milieu];
  (* fusion des deux sous vecteurs *)
  Fusion(T1,T2,n1,n2,T3,nb);
  (* mise à jour du vecteur T *)
   Pour i=0(1)nb-1 faire
        T[i + inf] ← T3[i+1];
Fin (* Fusion1 *);
```

Tri Fusion d'un tableau de taille N

- **Tâche importante: fusionner les sous parties triées T[inf..milieu] et T[milieu+1..sup] pour obtenir la partie triée T[inf..sup] du vecteur T[1..Nb_elts].**
- **On fait une adaptation de l'algorithme précédent:**

```
Procedure Fusion2(Var T:tableau, inf, sup,
  milieu)
(*Objectif: fusionner les sous parties
  triées T[inf..milieu] et T[milieu+1..sup]
  pour obtenir la partie triée T[inf..sup] du
  vecteur T[1..Nb_elts].
Paramètres entrants:
  T: tableau;
  1<=inf<=milieu<sup<=Nb_elts: bornes des
  sous parteir à fusionner;
Paramètres sortants: T, tableau avec la
  partie T[inf...sup] triée*)
```

Tri Fusion d'un tableau de taille N

```
Variable locale:
  i, j, k: entier;
  aux : tableau;
Début (* Fusion2 *)
   k ← inf; i ← inf; j ← milieu + 1;
  Tant que (i ≤ milieu) et (j ≤ sup) faire
        début
              Si T[i] ≤ T[j] alors
                    début
                          aux[k] ← T[i];
                          i ← i+1;
                    fin
              Sinon
                    début
                          aux[k] ← T[j];
                          j ← j+1;
                    fin
               k ← k+1;
        fin
```

Tri Fusion d'un tableau de taille N

```
Tant que (i ≤ milieu) faire
    début
        aux[k] ← T[i];
        i ← i+1;
        k ← k+1
    fin

Tant que (j ≤ sup) faire
    début
        aux[k] ← T[j];
        j ← j+1;
        k ← k+1
    fin
Pour i = inf (1) sup faire
    T[i] ← aux[i];
Fin (* Fusion2 *)
```

Tri Fusion d'un tableau de taille N

```
Procédure TriFusion (Var T : tableau,
  inf,sup :entier);
(* Objectif: trier la partie T[inf..sup] du
  tableau T, par ordre croissant en
  utilisant l'algorithme de tri fusion;
  Paramètres entrants:
  T: tableau non trié;
  inf<sup: borne de la partie à trier;
  Paramètres sortants: T, tableau avec la
  partie T[inf..sup] triée*)
Variable locale:
  milieu : entier;
```

Tri Fusion d'un tableau de taille N

```
TriFusion(T,inf, sup)
    Début
        Si (inf < sup) alors
            milieu ← div(inf+sup,2);
            TriFusion (T,inf,milieu);
            TriFusion(T,milieu+1,sup);
            Fusion1(T,inf,sup,milieu);
        fSi
    Fin
```

Tri Fusion d'un tableau de taille N

- La fusion de deux tableaux triés de taille k se fait en $O(k)$ comparaison.
- On considère les notations suivantes:

$$\lfloor n/2 \rfloor = \begin{cases} E(\frac{n}{2}) & si\ n\ est\ impair \\ \frac{n}{2} & sinon \end{cases}$$

et

$$\lceil n/2 \rceil = \begin{cases} E(\frac{n}{2}) + 1 & si\ n\ est\ impair \\ \frac{n}{2} & sinon \end{cases}$$

on a:

$$n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$$

Ainsi, pour l'algorithme de tri fusion, on a:

$$C(1) = O(1)\ \ et\ \ C(N) = C\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + C\left(\left\lceil \frac{N}{2} \right\rceil\right) + O(N);\ N > 1$$

Tri Fusion d'un tableau de taille N

On suppose sans nuire à la généralité que $N = 2^k$ $(k = \log_2(N))$. En remplaçant $O(N)$ par $c.N$ $(c > 0)$, on obtient:

$$C(N) = 2C\left(\frac{N}{2}\right) + c.N = 4C\left(\frac{N}{4}\right) + 2c.N$$

$$= 8C\left(\frac{N}{8}\right) + 3c.N = ... = 2^jC\left(\frac{N}{2^j}\right) + jc.N$$

$$= ... = 2^kC\left(\frac{N}{2^k}\right) + kc.N = NC(1) + cN\log_2(N)$$

D'où

$$C(N) = O(N\log_2(N))$$

Tri Rapide d'un tableau de taille N

- **Principe**: Il consiste à chaque étape à choisir un élément particulier du tableau appelé pivot, et le positionner à sa place en partitionnant le tableau en trois parties: les éléments plus petits que le pivot, le pivot et les éléments plus grands que le pivot. Dès que le pivot est positionné, on répète l'opération d'une par sur le sous tableau de gauche, d'autre par sur le sous tableau à droite.

- Le premier élément de la partie à trié est généralement choisi comme pivot. Cependant, on peut faire d'autres choix du pivot, en fonction des information disponible sur le tableau à trier

Tri Rapide d'un tableau de taille N

- **Exemple:**

i=1	40	19	18	11	28	14	25	7
2	19	18	11	28	14	25	7	40
3	18	11	7	14	19	25	28	40
4	11	7	14	18	19	25	28	40
5	7	11	14	18	19	25	28	40
6	7	11	14	18	19	25	28	40
7	7	11	14	18	19	25	28	40
8	7	11	14	18	19	25	28	40
	7	11	14	18	19	25	28	40

: Pivot

Tri Rapide d'un tableau de taille N

```
Procédure TriRapide (Var T : tableau,
  IDeb,IFin :entier);
(* Objectif: trier la partie T[IDeb..IFin]
  du tableau T, par ordre croissant en
  utilisant l'algorithme de tri Rapide;
  Paramètres entrants:
  T: tableau non trié;
  inf<sup: borne de la partie à trier;
  Paramètres sortants: T, tableau avec la
  partie T[IDeb ..IFin] triée*)
Variable locale:
  IGauche, IDroite: entier;
  Pivot : réel;
```

Tri Rapide d'un tableau de taille N

Début

(* Initialisation *)
$IGauche \leftarrow Ideb + 1;$
$IDroite \leftarrow IFin;$
$Pivot \leftarrow T[IDeb];$
(* Partition du Tableau *);
Tant que $(IGauche \leq IDroite\ )$ **faire**
début
Tant que $(IGauche \leq IFin)$ **et** $(T[IGauche] < Pivot)$ **faire**
début
$T[IGauche - 1] \leftarrow T[IGauche];$
$IGauche \leftarrow IGauche + 1;$
fin
Tant que $(Idroite \geq Ideb)$ **et** $(T[IDroite] \geq Pivot)$ **faire**
début
$IDroite \leftarrow IDroite - 1;$
fin

Si $(Idroite \geq Ideb)$ **et** $(IGauche \leq Idroite)$ **alors**
début
$Echange(T[Igauche]\, T[IDroite]);$
$IIDroite \leftarrow IDroite - 1;$
fin
fin
$T[Igauche - 1] \leftarrow Pivot;$
Si $(IDeb < Igauche - 1)$ **alors**
début
$TriRapide(T, IDeb,\ IGauche - 2);$
fin
Si $(IFin > IGauche)$ **alors**
début
$TriRapide(T,\ IGauche,\ IFin);$
fin
Fin

Tri Rapide d'un tableau de taille N

- La partition d'un tableau de taille k se fait en $c.k$ opérations où c est une constante.
- Soit $C(N)$ le nombre d'opérations effectuées pour le tri du tableau de taille N par la méthode du tri rapide. On a:

$$C(N) = C_i(N) = \begin{cases} O(1) & si\ N = 1 \\ C(i-1) + C(N-i) + c.N & si\ N > 1 \end{cases}$$

où i désigne la position du pivot dans le tableau.

- **Cas favorable**: Le pivot se trouve au milieu de la partie à trier. Alors

$$C(N) = 2C\left(\frac{N}{2}\right) + cN$$

D'où

$$C(N) \approx O(N \log N)$$

Tri Rapide d'un tableau de taille N

- **Cas défavorable**: Le tableau est trié. Alors

$$C(N) = C(N-1) + cN$$

D'où

$$C(N) = O(N^2)$$

- **Cas moyen**: Le pivot peut se trouver à n'importe qu'elle position dans le tableau et la probabilité pour qu'i se trouve à la position i est de $\frac{1}{n}$. Ainsi, en prenant la moyenne des nombres d'opérations $C_i(N)$ on a:

$$C(N) = \frac{1}{N}\sum_{i=1}^{N} C_i(N) = cN + \frac{1}{N}\left(\sum_{i=1}^{N} C(i-1) + \sum_{i=1}^{N} C(N-i)\right)$$

En multipliant membre à membre par N, on obtient:

$$NC(N) = cN^2 + 2\sum_{i=1}^{N} C(i-1)$$

De même pour $N > 1$, on a:

$$(N-1)C(N-1) = c(N-1)^2 + 2\sum_{i=1}^{N-1} C(i-1).$$

En soustrayant membre à membre, on a:

$$NC(N) - (N-1)C(N-1) = c(2N-1) + 2C(N-1)$$

Ainsi

$$NC(N) - (N+1)C(N-1) = c(2N-1).$$

D'où

$$\frac{NC(N) - (N+1)C(N)}{(N+1)N} = \frac{c(2N-1)}{(N+1)N}$$

et

$$\frac{(C(N)}{N+1} - \frac{C(N-1)}{(N)} = c\left(\frac{3}{N+1} - \frac{1}{N}\right).$$

De proche en proche, on montre que pour $N \geq 3$,

$$\frac{C(N)}{N+1} - \frac{C(1)}{2} = c\sum_{k=3}^{N}\frac{1}{k} + c\left(\frac{3}{N+1} - \frac{1}{2}\right).$$

Comme

$$\log(N) \leq \sum_{k=1}^{N}\frac{1}{k} \leq \log(N+1),$$

On déduit que

$$C(N) = O(N\log N)$$

Résolution de certaines formules de récurrence

On considère la relation de récurrence suivante:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

où $a \geq 1$, $b > 1$ et $f(n)$ est une fonction positive à partir d'un certain rang. La relation (1) donne le coût d'un algorithme qui résout de manière récursive un problème de taille n en le subdivisant en a sous problème de taille n/b. La fonction $f(n)$ comporte le coût de la subdivision et de la reconstruction.
Le théorème suivant donne une méthode générale pour déterminer l'ordre de grandeur de $T(n)$

Théorème Général (Master Theorem)

Soient $a \geq 1$, $b > 1$ et $f(n)$ une fonction définie sur $\mathbb{N}$. Soit $T(n)$ la fonction définie par la récurrence suivante

$$T(n) = aT(n/b) + f(n) \tag{2}$$

où le terme n/b signifie soit $\lfloor n/b \rfloor$, soit $\lceil n/b \rceil$. Alors $T(n)$ possède les bornes asymptotiques suivantes:
1) Si $\exists \varepsilon > 0$ tel que $f(n) = O(n^{\log_b(a)-\varepsilon})$, alors $T(n) = \Theta(n^{\log_b(a)})$.
2) Si $f(n) = \Theta(n^{\log_b(a)})$, alors $T(n) = \Theta(n^{\log_b(a)} \ln n)$.
3) si $\exists \varepsilon > 0$ et $\exists c < 1$ tels que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ et $af(n/b) \leq cf(n)$ pour n assez grand, alors $T(n) = \Theta(f(n))$.

Exemple:

$$T(n) = 9T(n/3) + n, \quad T(n) = T(2n/3) + 1, \quad T(n) = 3T(n/4) + n \ln n$$

Outline

Calcul approché des sommes des séries numériques convergentes

On considère la série numérique

$$\sum_{n \geq n_0} u_n.$$

On suppose que cette série est convergente.

Problème:

Déterminer la somme

$$S = \sum_{n=n_0}^{+\infty} u_n$$

de cette série.

Sauf pour les cas rares dans la pratique, il est difficile de déterminer la valeur exacte de la somme S. On détermine généralement une approximation de cette somme par un calcul approché.

- Étant donné un réel $\epsilon > 0$, le problème de calcul de la valeur approchée de la somme S de la série convergente

$$\sum_{n \geq n_0} u_n.$$

consiste à déterminer le rang $n \geq n_0$ tel que

$$|R_n| \leq \epsilon$$

où R_n est le reste d'ordre n de la série. La valeur approchée de S à ϵ près est alors donnée par la somme partielle d'ordre n de la série:

$$S_n = \sum_{k=n_0}^{n} u_k$$

. en effet, on a:

$$|S - S_n| = \left| \sum_{k=n+1}^{\infty} u_k \right| = |R_n| \leq \epsilon.$$

- De façon, pratique on cherche à majorer $|R_n|$ par une suite plus "simple" ρ_n vérifiant les propriétés suivantes:
  1. $\forall n \geq n_0$, $|R_n| \leq \rho_n$;
  2. $(\rho_n \to 0)$ quand $(n \to +\infty)$ assez rapidement;
  3. $\forall n \geq n_0$, ρ_n est "facilement" calculable.

  Ainsi,
  $$\rho_n \leq \epsilon \Rightarrow |R_n| \leq \epsilon.$$
- On doit chercher à majorer $|R_n|$ le plus finement possible (éviter le plus possible des majorations grossières).
- En général, la détermination de la suite ρ_n (par suite du rang n) dépend du terme générale de la série et est basée sur le critère de convergence utilisé.

Calcul de la valeur approchée de la série

Après détermination de la suite ρ_n, le calcul de la valeur approchée de S se fait de manière itérative. Deux cas de figure peuvent se présenter:

- Si l'expression de ρ_n est assez "simple" et permet de calculer facilement la valeur de n tel que $|\rho_n| \leq \epsilon$, alors on détermine directement S_n. On peut par exemple utiliser le schéma d'algorithme suivant:

Donnée: $n0,\ n$;
$S \leftarrow U(n0)$;
Pour $i = n0 + 1(1)n$ **faire**
$S \leftarrow S + U(i)$;
Fpour

Calcul de la valeur approchée de la série

- Dans le cas contraire, on détermine à partir du rang n_0, les approximation successives de S, jusqu'à la réalisation de la condition $|\rho_n| \leq \epsilon$. On peut par exemple utiliser le schéma d'algorithme suivant:

Donnée: $n0,\ \epsilon$;
$S \leftarrow U(n0)$;
$i \leftarrow n0 + 1$;
Tant que $\rho(i) > \epsilon$ **faire**
$S \leftarrow S + U(i)$;
$i + 1$;
FTantque

- Lorsque la série de terme général (u_n) vérifie les hypothèses du critère des séries alternées, on a la majoration suivante du reste d'ordre n:

$$|R_n| \leq |u_{n+1}|$$

Ainsi, on prend $\rho_n = |u_{n+1}|$

Exemple:

$$S = \sum_{n=1}^{+\infty} \frac{(-1)^{-n}}{n} \text{ à } 10^{-3}\text{près}$$

Il suffit que $\frac{1}{n+1} \leq 10^{-3}$, ie $n > 999$. Ainsi, S_{999} donne une valeur approchée de S à 10^{-3} près.

- Lorsque la série de terme général (u_n) vérifie les hypothèses du critère des séries trigonométrique, on a la majoration suivante du reste d'ordre N:

$$|R_N| \leq \frac{|u_{N+1}|}{|\sin(\theta/2)|}$$

Ainsi, on prend $\rho_n = |u_{n+1}|$

Exemple:

$$S = \sum_{n=1}^{+\infty} \frac{\cos(n/10)}{\sqrt{n}} \text{ à } 10^{-5} \text{près}$$

Si la série de terme générale u_n converge par la règle de Cauchy, alors on a:

$$\lim_{n \leftarrow +\infty} \sqrt[n]{|u_n|} = L < 1$$

D'où, pour toute valeur λ telle que $L < \lambda < 1$, il existe $N_\lambda \in \mathbb{N}$ tel que

$$n > N_\lambda \Rightarrow \sqrt[n]{u_n} \leq \lambda.$$

Il existe donc un rang N et un réel r_N tel que

$$n > N \Rightarrow |u_n| \leq r_N^n$$

On obtient alors la majoration suivante de $|R_N|$:

$$|R_N| \leq \frac{(r_N)^{(N+1)}}{1 - r_N}.$$

Ainsi, on peut poser

$$\rho_N = \frac{(r_N)^{(N+1)}}{1 - r_N}.$$

Exemple:

$$S = \sum_{n=1}^{+\infty} \frac{1}{n^n} \text{ à } 10^{-7} \text{ près}$$

Si la série de terme générale u_n converge par la règle de d'Alembert, alors on a:

$$\lim_{n \leftarrow +\infty} \left| \frac{u_{n+1}}{u_n} \right| = L < 1$$

D'où, pour toute valeur λ telle que $L < \lambda < 1$, il existe $N_\lambda \in \mathbb{N}$ tel que

$$n > N_\lambda \Rightarrow \left| \frac{u_{n+1}}{u_n} \right| \leq \lambda.$$

Il existe donc un rang N et un réel r_N tel que

$$n > N \Rightarrow |u_{n+1}| \leq r_N |u_n|$$

On obtient alors la majoration suivante de $|R_N|$:

$$|R_N| \leq \frac{|u_{(N+1)}|}{1 - r_N}.$$

Ainsi, on peut poser

$$\rho_N = \frac{|u_{(N+1)}|}{1 - r_N}.$$

Exemple:

$$S = \sum_{n=1}^{+\infty} \frac{n! n^n}{(2n)!} \text{ à } 5.10^{-10} \text{ près}$$

Application au calcul de certaines intégrales

Soit f une fonction bornée sur $[a,b] \subset \mathbb{R}$. on désire déterminer une approximation à ϵ près de l'intégrale

$$I = \int_a^b f(x)dx.$$

On suppose que:

1. **f est la somme d'une série de fonctions de terme général $(f_n)_{n \geq n_0}$ définie sur $[a,b]$:**

$$\forall x \in [a,b],\ f(x) = \sum_{n=n_0}^{+\infty} f_n(x).$$

2. **Les fonctions f_n ($n \geq n_0$ sont intégrables sur $[a,b]$.**
3. **La série de fonction $\sum_{n \geq n_0} f_n$ converge uniformément sur $[a,b]$ vers f.**

Alors d'après le Théorème d'Intégration Terme à Terme, on a:

$$I = \int_a^b f(x)dx = \int_a^b (\sum_{n=n_0}^{+\infty} f_n(x))dx = \sum_{n=n_0}^{+\infty} \int_a^b f_n(x)dx = \sum_{n=n_0}^{+\infty} a_n = A,$$

où

$$a_n = \int_a^b f_n(x)dx.$$

Ainsi, déterminer une approximation de I à ϵ près, revient à déterminer une approximation de A à ϵ près.

Développement de Taylor

Soit $f \in \mathcal{C}^n([a,b])$. On suppose que $f^{(n+1)}$ existe sur $]a,b[$. Soit $x_0 \in [a,b]$. Alors pour tout $x \in [a,b]$, il existe ξ_x compris entre x et x_0 tel que

$$f(x) = \underbrace{f(x_0) + \sum_{k=1}^{n} f^{(k)}(x_0)\frac{(x-x_0)^k}{k!}}_{P_n f(x)} + \underbrace{f^{(n+1)}(\xi_x)\frac{(x-x_0)^{n+1}}{(n+1)!}}_{R_n f(x)} \quad (3)$$

- La relation (3) définit le développement de Taylor à l'ordre n de la fonction f au voisinage de x_0 sur $[a,b]$.
- $P_n f$ est la partie principale du développement de Taylor (polynôme de Taylor) à l'ordre n de la fonction f au voisinage de x_0 sur $[a,b]$.
- La fonction $R_n f$ définit le reste d'ordre n du développement de Taylor de la fonction f au voisinage de x_0 sur $[a,b]$.
- Lorsque $x_0 = 0$, on obtient alors le développement de McLaurin de la fonction f.

Notion d'Erreur d'Approximation.

Soit $x_0 \in [a, b]$ fixé. On suppose que la fonction f admet un développement de Taylor à l'ordre n au voisinage de x_0 sur $[a, b]$. On effectue l'approximation suivante

$$f(x) \approx P_n f(x), \qquad x \in]a, b[\tag{4}$$

On a

$$f(x) - P_n f(x) = R_n f(x) = f^{(n+1)}(\xi_x) \frac{(x - x_0)^{n+1}}{(n+1)!}, \tag{5}$$

où $\xi_x \in (x, x_0)$. Ainsi,

$$|f(x) - P_n f(x)| = \left| f^{(n+1)}(\xi_x) \frac{(x - x_0)^{n+1}}{(n+1)!} \right| \tag{6}$$

Si $f^{(n+1)}$ est bornée sur $]a, b[$, alors

$$\exists M_{n+1} > 0, \quad |f^{(n+1)}(\xi_x)| \leq M_{n+1}.$$

Notion Erreur d'Approximation (Suite).

On a la majoration suivante de l'erreur d'approximation:

$$|f(x) - P_n f(x)| \leq M_{n+1} \left| \frac{(x - x_0)^{n+1}}{(n+1)!} \right| \leq M_{n+1} \frac{(b-a)^{n+1}}{(n+1)!} \tag{7}$$

En posant $h = x - x_0$, on a:

$$E_n(h) = f(x) - P_n f(x) = f^{(n+1)}(\xi_x) \frac{h^{n+1}}{(n+1)!}. \tag{8}$$

Si $f^{(n+1)}$ est bornée sur $]a, b[$,

$$E_n(h) = O\left(h^{n+1}\right) \tag{9}$$

On dit alors que l'approximation de $f(x)$ par $P_n f(x)$ est d'ordre $n+1$ au voisinage de x_0.

Exemple 1: On considère la fonction définie sur $\mathbb{R}$ par $f(x) = \cos(x)$.

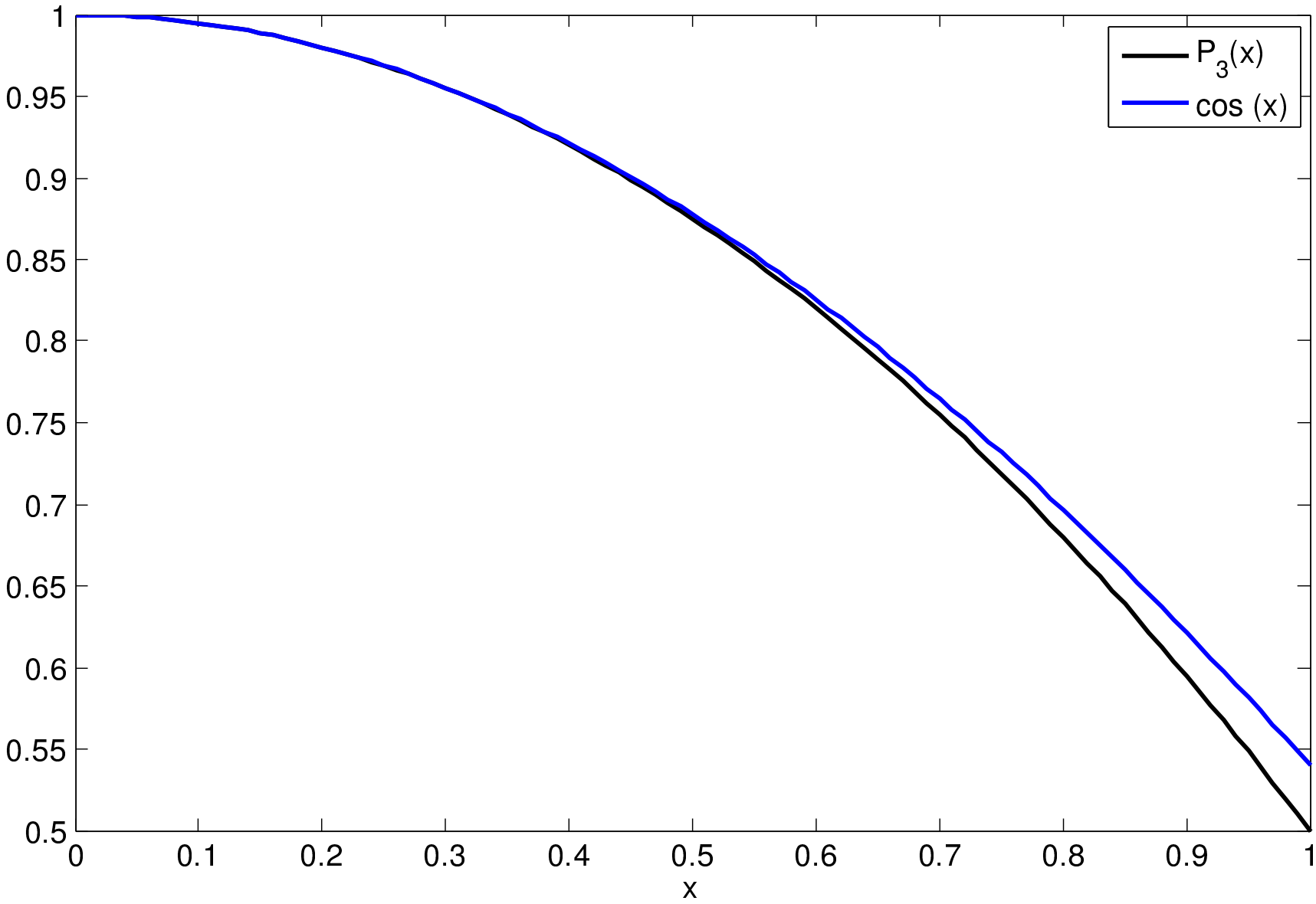
1. Donner une approximation de f au voisinage de 0 pour $n = 3$.
2. Utiliser cette approximation pour obtenir des valeurs approchées de $\cos(0.01),\ \cos(0.05),\ \cos(0.1),\ \cos(1)$.
3. Donner une majoration de l'erreur d'approximation dans chaque cas.

Au voisinage de 0 on a en utilisant le développement de Taylor à l'ordre 3 de la fonction $\cos x$:

$$\cos x \approx 1 - \frac{x^2}{2}$$

Sur l'intervalle $[0, 1]$, la majoration de la dérivée d'ordre 4 de cette fonction est $M_4 = 1$. D'où, l'erreur d'approximation est majorée par $E_4 = \frac{1}{24} \approx 0.0416666$.

x	$P_3(x)$	$\cos x$	Erreur
0.01	0.99995	0.999950000416665	4.1666257851889e-10
0.05	0.99875	0.998750260394966	2.60394966256072e-7
0.1	0.995	0.995004165278026	4.16527802571398e-6
1	0.5	0.540302305868140	0.040302305868139 8


1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
x
P3(x)
cos (x)

Exemple 2: On considère la fonction définie sur $\mathbb{R}_+^*$ par $f(x) = \ln(x)$.

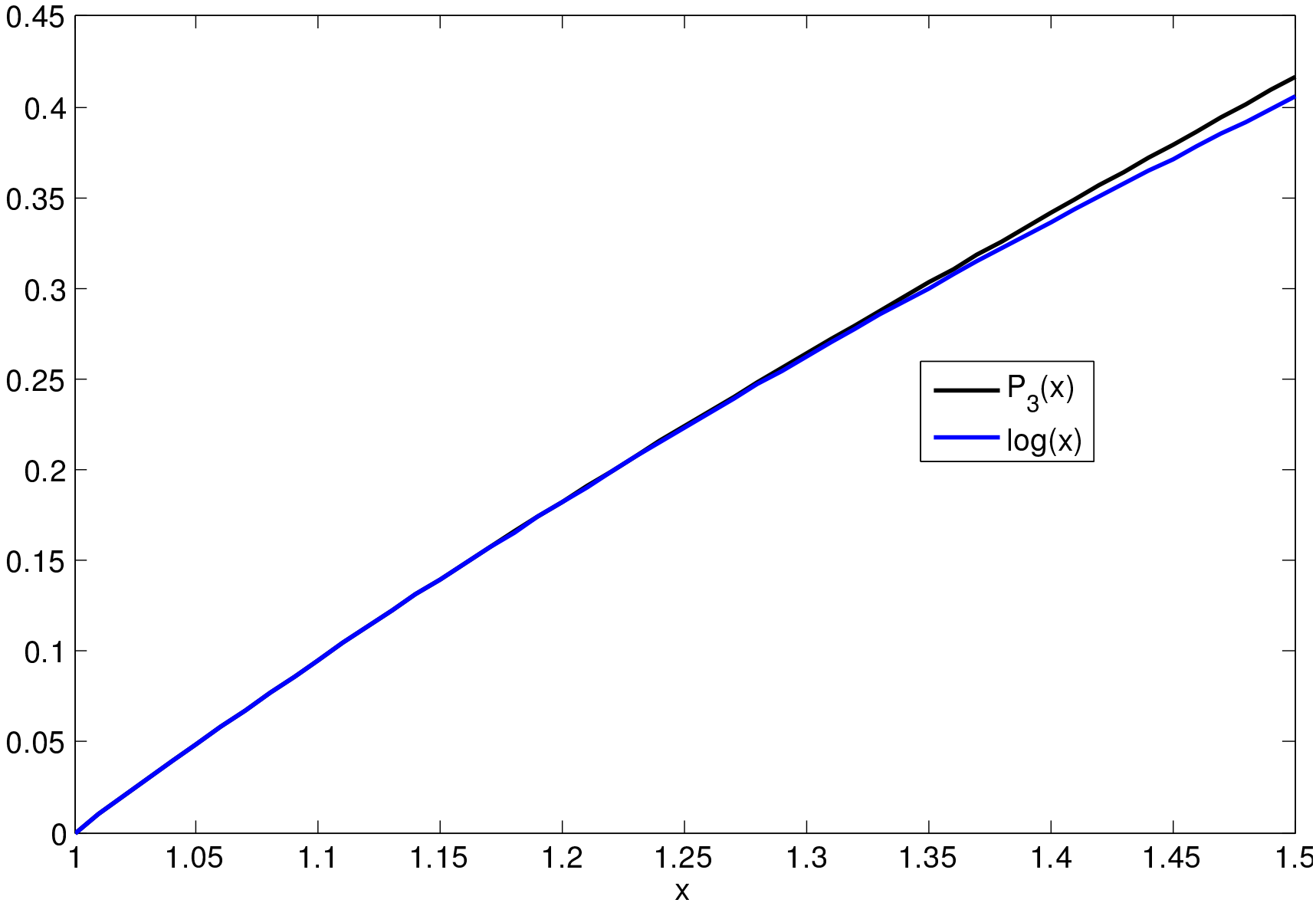
1. Donner une approximation de f au voisinage de 1 pour $n = 3$.
2. Utiliser cette approximation pour obtenir des valeurs approchées de $\ln(1.1)$, $\ln(1.2)$, $\ln(1.3)$, $\ln(1.4)$, $\ln(1.5)$.
3. Donner une majoration de l'erreur d'approximation dans chaque cas.

Au voisinage de 0 on a en utilisant le développement de Taylor à l'ordre 3 de la fonction $\cos x$:

$$\ln x \approx (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3}$$

Sur l'intervalle $[1, 1.5]$, la majoration de la dérivée d'ordre 4 de cette fonction est $M_4 = 6$. D'où, l'erreur d'approximation est majorée par $E_4 = \frac{1}{64} \approx 0.015625$.

x	$P_3(x)$	$\ln x$	Erreur
1.1	0.095333	0.09531	-2.3154e-5
1.2	0.18267	0.18232	-0.00034511
1.3	0.264	0.26236	-0.0016357
1.4	0.34133	0.33647	-0.0048611
1.5	0.41667	0.40547	-0.011202


0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1
1.05
1.1
1.15
1.2
1.25
1.3
1.35
1.4
1.45
1.5
x
P3(x)
log(x)

Applications: exp(x), cos(x), sin(x)

Théorème 1:

Soit $f : \mathbb{R} \to \mathbb{R}_+$ une fonction vérifiant $f' = f$ et $f(0) = 1$. Alors $\forall x \in \mathbb{R}$,

$$f(x) = \sum_{k=0}^{+\infty} \frac{x^k}{k!} = \exp(x).$$

Théorème 2:

Soient $f, g : \mathbb{R} \to [-1, 1]$ deux fonctions vérifiant $f' = g$ et $g' = -f$ ainsi que $f(0) = 0$ et $g(0) = 1$. Alors $\forall x \in \mathbb{R}$,

$$f(x) = \sum_{k=0}^{+\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} = \sin(x) \; et \; g(x) = \sum_{k=0}^{+\infty} \frac{(-1)^k}{(2k)!} x^{2k} = \cos(x).$$

Définition

On appelle série entière, toute série de fonction de la forme:

$$\sum_{n\geq 0} a_n(z-z_0)^n = \sum_{n\geq 0} a_n Z^n$$

où (a_n) est une suite de scalaire et $Z = z - z_0$.

Rayon de convergence

On appelle rayon de convergence de la série entière $\sum_{n\geq n_0} a_n Z^n$, le réel positif R_c vérifiant les propriétés suivantes:

- **$\sum_{n\geq 0} a_n Z^n$ converge pour $|Z| < R_c$;**
- **$\sum_{n\geq 0} a_n Z^n$ diverge pour $|Z| > R_c$;**

Le rayon de convergence de la série entière $\sum_{n\geq n_0} a_n Z^n$ est déterminé par la formule d'Hadamard suivante:

$$\frac{1}{R_c} = \limsup_{n\to+\infty} \sqrt[n]{|a_n|}$$

Rayon de convergence: Critère de d'Alembert

Si la série entière $\sum_{n\geq 0} a_n Z^n$ est telle que pour n assez grand on a $a_n \neq 0$ et si la suite $(|a_{n+1}|/|a_n|)$ admet une limite $L \in [0, +\infty]$ alors son rayon de convergence est donné par :

$$R_c = \frac{1}{L}$$

Exemple

$$\sum \frac{z^k}{k!}; \sum \frac{(-1)^k}{(2k+1)!} z^{2k+1}; \sum k! z^k$$

Le disque centré en z_0 et de rayon $R_c > 0$

$$D(z_0, R_c) = \{z \in C, |Z| = |z - z_0| < R_c\}$$

est appelé disque ouvert de convergence de la série $\sum_{n\geq 0} a_n Z^n$.

Soit La série entière $\sum_{n\geq 0} a_n Z^n$ de rayon de convergence $R_c > 0$.

- **$\sum_{n\geq 0} a_n Z^n$ converge absolument sur son disque ouvert de convergence $D(z_0, R_c)$.**
- **Soit r un réel tel que $0 < r < R_c$. La série $\sum_{n\geq 0} a_n Z^n$ converge normalement sur le disque $D(z_0, r)$**

Ainsi, on peut définir sur $D(z_0, R_c)$ la fonction

$$f(z) = \sum_{k=0}^{+\infty} a_k (z - z_0)^k$$

.

- **La fonction f est dérivable à tous les ordres sur $D(z_0, R_c)$ et on a:**

$$f^{(k)}(z) = \sum_{n=0}^{+\infty} \frac{(n+k)!}{n!} a_{n+k} (z - z_0)^n = \sum_{n=k}^{+\infty} \frac{n!}{(n-k)!} a_n (z - z_0)^{(n-k)}$$

Applications: Implémenter exp(x)

Objectif: Approcher $f(x) = \exp(x)$ à 10^{-20} près avec $x \in [0, 1]$.
Méthode: Pour tout $x \in]-11[$, on a En utilisant le développement de Taylor de la fonction f au voisinage de $x_0 = 0$, on a l'approximation suivante:

$$f(x) = P_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}.$$

L'erreur d'approximation est donnée par:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}, \xi \in (x_0, x).$$

Ainsi, on a la majoration suivante:

$$|R_n(x)| = \frac{\exp(\xi)}{(n+1)!}|x|^{n+1} \leq \frac{3}{(n+1)!}.$$

Il suffit de déterminer n tel que $\frac{3}{(n+1)!} \leq 10^{-20}$. De proche en proche, on trouve que $n = 21$ convient.
Pour $x \in [0, 1]$ l'approximation de $\exp(x)$ est déterminer en appliquant la méthode de Horner à $P_n(x)$.

Implémentation:Pour $x \in [0,1]$ l'approximation de $\exp(x)$ est déterminer en appliquant la méthode de Horner à $P_n(x)$.

$$\sum_{k=0}^{n} \frac{x^k}{k!} = \left(\left(\cdots\left(\left(\left(\frac{x}{n}+1\right)\frac{x}{n-1}+1\right)\frac{x}{n-2}+1\right)\cdots\right)\frac{x}{1}+1\right)$$

Considérons maintenant le cas ou on souhaiterai obtenir une approximation de $\exp(x)$ avec une erreur relative de $\varepsilon = 10^{-14}$, pour $x \in [-10^6, 10^6$.
Ceci revient a determiner n tel que $|R_n(x)|/\exp(x) < \varepsilon$ ie

$$\frac{|x|^{n+1}}{(n+1)!} < \varepsilon$$

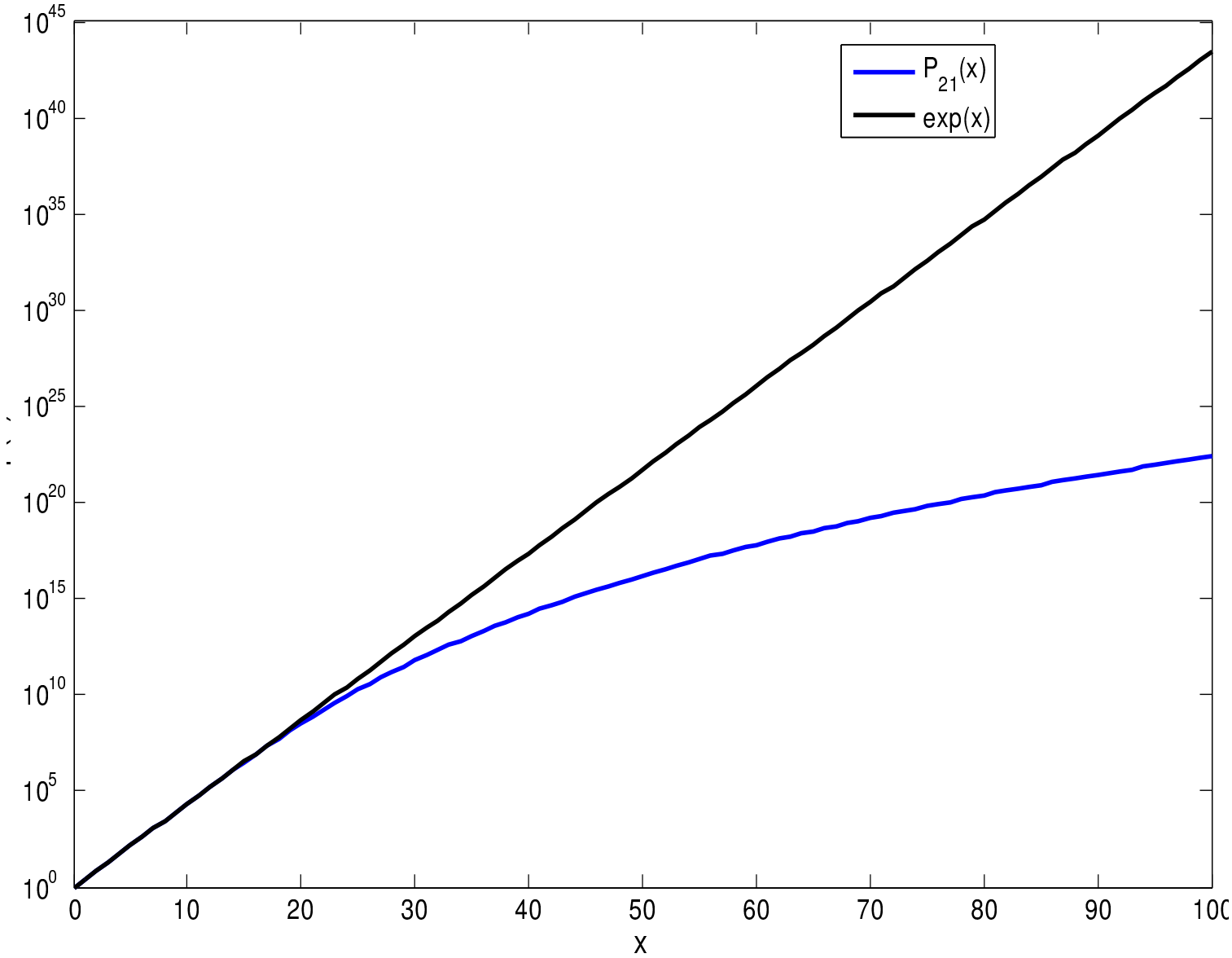
Pour $x = 10^6$, il faudra prendre $n \gg 10^6$. Ce qui n'est pas pratique.

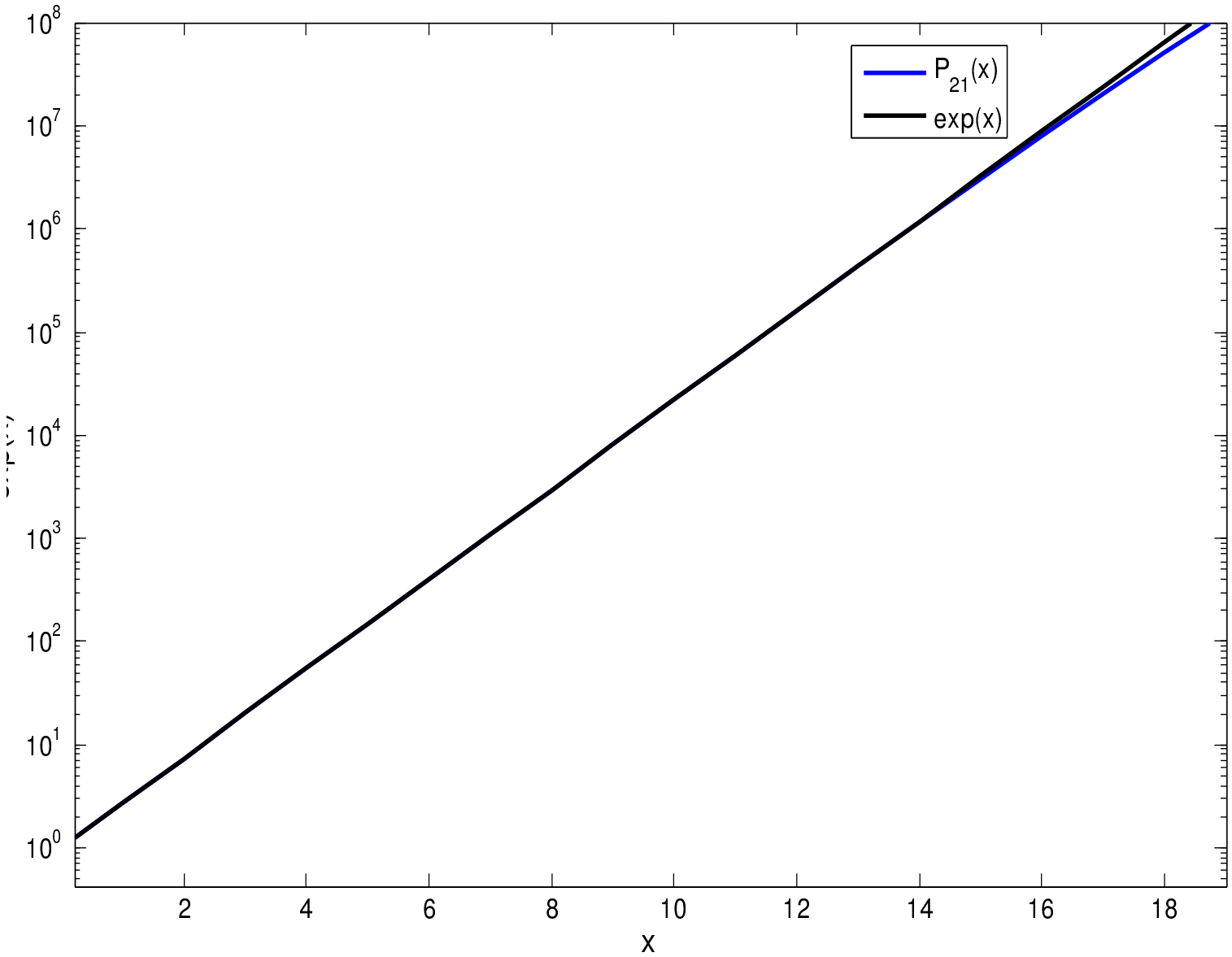
Applications: Implémenter $\exp(x)$

Une alternative serait d'utiliser l'approche suivante (Reduction de l'argument):

1. Pour $x \in [0, 1]$, utiliser le développement de Taylor pour calculer une valeur approchée de $\exp(x)$ avec une precision de 10^{-20};
2. Pour $x \in]1, 2]$, trouver $t \in [1/2, 1]$ tel que $x = 2t$, puis calculer $\exp(x) = \exp(t)^2$;
3. Pour $x \in]1, 10^6]$, trouver $t \in [1/2, 1]$ et $k \in \{1, 2, \cdots, 20\}$ tels que $x = t2^k$, puis calculer $\exp(x) = [\exp(t)]^{2^k}$;
4. Pour $x < 0$, utiliser la formule $\exp(x) = \frac{1}{\exp(-x)}$.

Le calcul de $[\exp(t)]^{2^k}$ revient a élever k fois au carré. Chaque fois l'erreur relative est multipliée par 2, donc tout au plus par $2^{20} = 1048576 \approx 10^6$


10^{45}
10^{40}
10^{35}
10^{30}
10^{25}
10^{20}
10^{15}
10^{10}
10^{5}
10^{0}
0
10
20
30
40
50
60
70
80
90
100
X
$P_{21}(x)$
exp(x)


$P_{21}(x)$
exp(x)
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
2
4
6
8
10
12
14
16
18
x

Applications: Implémenter $\ln(x)$

Objectif: Approcher $f(x) = \ln(1+x)$ où $x \in [-\frac{1}{2}, \frac{1}{2}]$ à 10^{-20} près
Méthode: En utilisant le développement de Taylor de la fonction f au voisinage de $x_0 = 0$.
Pour tout $x \in]-1, 1[$, on a

$$f(x) = \sum_{k=1}^{+\infty} \frac{(-1)^{k+1}}{k} x^k.$$

Ainsi, une approximation de $f(x)$ peut être donnée par:

$$P_n(x) = \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k} x^k.$$

L'erreur d'approximation est donnée par:

$$R_n(x) = \sum_{k=n+1}^{+\infty} \frac{(-1)^{k+1}}{k} x^k.$$

Applications: Implémenter $\ln(x)$

Ainsi, on a la majoration suivante:

$$|R_n(x)| = \left| \sum_{k=n+1}^{+\infty} \frac{(-1)^{k+1}}{k} x^k \right| \leq \sum_{k=n+1}^{+\infty} \frac{|x^k|}{k}.$$

Ainsi, pour $|x| \leq \frac{1}{2}$, On a

$$|R_n(x)| \leq \sum_{k=n+1}^{+\infty} \left(\frac{1}{2}\right)^k = 2^{-n}.$$

Il suffit de déterminer n tel que $2^{-n} \leq 10^{-20}$. De proche en proche, on trouve que $n = 67$ convient.

Remarque: Pour $|x|$ proche de 1, cette méthode devient lente. Par exemple, si $x = 1$, on $\sum_{k=1}^{+\infty} \frac{(-1)^{k+1}}{k}$ converge vers $\ln(2)$ tres lentement. Il faut 10^6 iterations pour obtenir une approximation de $\ln(2)$ a 10^{-6} pres.

Applications: Implémenter $\ln(x)$

Une autre approche consiste a utiliser le développement sur $]-1,1[$ de la fonction

$$g(t) = \ln\left(\frac{1+t}{1-t}\right) = \ln(1+t) - \ln(1-t) = \sum_{k=1}^{+\infty} \frac{(-1)^{k+1}}{k} t^k + \sum_{k=1}^{+\infty} \frac{t^k}{k}.$$

Ainsi, on

$$\ln\left(\frac{1+t}{1-t}\right) = 2\sum_{k=0}^{+\infty} \frac{t^{2k+1}}{2k+1}$$

La convergence de cette série est semblable a celle d'une série géométrique pour $|t| < 1$.

Exemple: Determiner n permettant d'obtenir une approximation de $\ln(2)$ a 10^{-20} pres.

Avec $t \in [0, \frac{1}{3}]$, on peut calculer assez rapidement une bonne approximation de $\ln(x)$ pour $x \in [1,2]$.

Applications: Implémenter $\ln(x)$

Plus généralement, on pourra utiliser l'approche de reduction de l'argument suivante pour determiner une bonne approximation de $\ln(x)$, $x > 0$.

- Pour $x \in [1, 2]$, trouver $t \in [0, \frac{1}{3}]$ tel que $x = \frac{1+t}{1-t}$, puis utiliser le développement precedent;
- Pour $x > 2$, trouver un entier k tel que $\frac{x}{2^k} \in [1, 2]$ puis determiner une approximation de $\ln(x) = \ln\left(\frac{x}{2^k}\right) + k\ln(2)$;
- Pour $0 < x < 1$, on applique la formule $\ln(x) = -\ln\left(\frac{1}{x}\right)$

Outline

A Title

Don’t mess with my text.

Outline

Introduction

- **MATLAB** = **MA**trix **LAB**oratory.
- Logiciel de Calcul Scientifique en développement depuis le début des années 80;
- Initialement conçu pour faciliter les opérations de calcul matriciel
- Est aujourd'hui un outils assez puissant de calcul numérique comportant plusieurs TOOLBOX
- Les objets de base de Matlab sont les matrices: ńăcomprend tout comme matrice et/ou tableauăż
- Un langage interprété

Environnement

- Lancement

Environnement

- Acceuil

Environnement

- Aide sous Matlab:

```
>> help  % affiche l'aide de Matlab
>> help ńămot cléăż  % affiche lŠaide Matlab à propos d
```

Outline

Outline

Outline

Outline

• On considère dans ce chapitre la résolution numérique du système

$$Ax = b, \tag{10}$$

où $A = (a_{ij})_{1\leq i,j\leq n} \in \mathbb{C}^{n\times n}$ et $b = (b_i)_{1\leq i\leq n} \in \mathbb{C}^n$ sont donnés, $x = (x_i)_{1\leq i\leq n} \in \mathbb{C}^n$ est le vecteur inconnu.

• La solution du SEL (10) joue un rôle important dans la resolution de plusieurs problèmes en sciences et en ingénierie.

• On distingue deux catégories de méthodes numériques pour la résolution des systèmes d'équations linéaires :

1. les **méthodes directes** qui permettent d'obtenir la solution exacte (lorsque l'influence des erreurs d'arrondi est négligeable) après un nombre fini d'opérations,
2. les **méthodes itératives** qui permettent d'obtenir à partir d'une approximation initiale, des approximations de plus en plus " proche " (dans le sens d'une norme donnée) de la solution exacte du système (lorsque la méthode converge). Dans ce cas on arrête le processus après un nombre maximal d'itération donné et/ou une certaine précision fixée.

• On s'interesse dans ce chapitre aux méthodes directes de résolution du SEL (10).
• On suppose dans la suite que la matrice A du SEL (10) est inversible ie $det(A) \neq 0$. On dit alors que le SEL (10) est de Cramer.
• Une méthode de résolution du SEL (10) est la méthode de Cramer qui procède comme suit:

$$x_i = \frac{\det(A_i)}{\det(A)}, \quad i = 1, 2, \cdots, n \tag{11}$$

où A_i est la matrice obtenue de A en remplaçant la colonne i de A par le vecteur b.

• **Coût de la méthode de Cramer:**
La méthode de Cramer nécessite le calcul de $n+1$ déterminant d'ordre n. or le cout de calcul d'un determinant d'ordre n est un $\mathbf{O(nn!)}$. D'où le coût de la méthode de Cramer est: $\mathbf{O((n+1)nn!)}$.

• Lorsque $n = 20$, en utilisant un ordinateur fonctionnant à $\mathbf{10^9\ flops}$, il faut environ $\mathbf{t = 1.2 \times 10^{12}s}$ soit environ $\mathbf{32420\ annees}$ pour résoudre le SEL (10).

Impossible d'utiliser la méthode de Cramer dans la pratique.

• Le principe des méthodes directes consiste à transformer le SEL (10) en un SEL facile a résoudre.

Systèmes Faciles à Résoudre

1- A est triangulaire

1.a- A est sup-triangulaire

Le SEL (10) est de la forme :

$$\begin{pmatrix} a_{11} & a_{12} & \cdot & \cdot & a_{1n} \\ & a_{22} & \cdot & \cdot & a_{2n} \\ & & \cdot & \cdot & \cdot \\ & & & a_{n-1n-1} & a_{n-1n} \\ & & & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \cdot \\ b_{n-1} \\ b_n \end{pmatrix} \qquad (12)$$

• La matrice A étant inversible, on a $a_{ii} \neq 0$ $(1 \leq i \leq n)$. Pour résoudre le système (12), on commence par calculer l'inconnu x_n; en remplaçant x_n par sa valeur dans l'équation $n-1$, On calcule x_{n-1}. On réitère le processus pour calculer $x_{n-2}, x_{n-3}, \cdots, x_2, x_1$.

En effet, on a:

$$x_n = \frac{b_n}{a_{nn}},$$

$$x_{n-1} = \frac{b_{n-1} - a_{n-1n}x_n}{a_{n-1n-1}},$$

$$x_{n-2} = \frac{b_{n-1} - a_{n-2n-1}x_{n-1} - a_{n-2n}x_n}{a_{n-2n-2}}$$

D'une manière générale, on a la **méthode de la Remontée** suivante:

$$\begin{cases} x_n = \frac{b_n}{a_{nn}} \\ x_k = \left(b_k - \sum_{j=k+1}^{n} a_{kj}x_j\right)/a_{kk}; \quad k = n-1, n-2, ..., 1 \end{cases} \tag{13}$$

1.b- A est Inf-triangulaire

En procédant comme précédemment, on obtient la **méthode de la Descente** suivante:

$$\begin{cases} x_1 = \frac{b_1}{a_{11}} \\ x_k = \left(b_k - \sum_{j=1}^{k-1} a_{kj}x_j\right)/a_{kk}; \quad k = 2, 3, ..., n \end{cases} \tag{14}$$

1.c- Coût des méthodes de la Remontée et de la Descente

Lemme

La résolution d'un système d'équations linéaires triangulaire se fait en n^2 opérations a virgule flottante.

2- $A = LU$ avec L Inf-triangulaire et U Sup-triangulaire

On a alors:

$$Ax = b \Longleftrightarrow LUx = b \Longleftrightarrow \begin{cases} Ly = b & (1) \\ Ux = y & (2) \end{cases}$$

On résout d'abord le système (1) par la méthode de la descente, puis on résout le système (2) par la méthode de la remontée.

- **Le coût de la méthode est $2n^2$**

3- $A = QR$ avec Q orthogonale ($Q^{-1} = Q^t$) et R Sup-triangulaire

On a alors:

$$Ax = b \Longleftrightarrow QRx = b \Longleftrightarrow \begin{cases} Qy = b & (1) \\ Rx = y & (2) \end{cases}$$

On détermine $y = Q^t b$, puis puis on résout le système (2) par la méthode de la remontée. Le calcul de y s'effectue comme suit: si $Q = (q_i j)$, alors on a

$$y_i = \sum_{j=1}^{n} q_{ji} b_j, i = 1, 2, \cdots, n$$

- **Le coût de la méthode est $3n^2 - n$**

Méthodes Directes

L'idée générale des méthodes directes de résolution du système

$$Ax = b \tag{15}$$

consiste à le transformer en un système équivalent de la forme

$$Bx = b' \tag{16}$$

"facile à résoudre" (où la matrice B est : diagonale, triangulaire, produit de matrices triangulaires ou d'une matrice triangulaire et d'une matrice orthogonale). La matrice B dépend de A et le vecteur b' dépend de b et où A. Les méthodes présentées ci-dessous décrivent les processus d'obtention de la matrice B et du vecteur b'.

Méthodes d'élimination de Gauss

La méthode directe de Gauss consiste à:

- transformer le système (15) en un système équivalent dont la matrice est triangulaire supérieure;
- Résoudre le système triangulaire est obtenu par la méthode de la remontée.

On a deux en deux grandes étapes :

1. la triangularisation du système qui s'opère en effectuant des opérations élémentaires (échange et addition) sur les linges de la matrice augmentée du système (15);
2. la résolution du système triangulaire obtenu.

On a le schéma d'algorithme suivant :

1^{ere} **étape** : *triangularisation du système.*

$$
\begin{aligned}
&A^{(1)} = A \\
&b^{(1)} = b \\
&k = 1, 2, ..., n-1 : \\
&\begin{cases}
a_{ij}^{(k+1)} = a_{ij}^{(k)}; \quad i = 1, ..., k; \quad j = 1, ...n \\
a_{ij}^{(k+1)} = 0; \quad i = k+1, ..., n; \quad j = 1, ...k \\
\textbf{Si } \ a_{kk}^{(k)} = 0, \ \ \textit{Rechercher et positionner le Pivot } \ \textbf{finSin} \\
a_{ij}^{(k+1)} = a_{ij}^{(k)} - \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)}; \quad i = k+1, ..., n; \quad j = k+1, ...n \\
b_i^{(k+1)} = b_i^{(k)}; \quad i = 1, ..., k \\
b_i^{(k+1)} = b_i^{(k)} - \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)}; \quad i = k+1, ..., n
\end{cases}
\end{aligned}
\tag{17}
$$

2^{eme} **étape** : *résolution du système triangulaire supérieure* :

$$
\begin{aligned}
& A^{(n)}x = b^{(n)} \\
& x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}} \\
& x_i = \frac{b_i^{(n)} - \sum_{j=i+1}^{n} a_{ij}^{(n)}}{a_{ii}^{(n)}} \quad i = n-1, ..., 1
\end{aligned}
\tag{18}
$$

Coût de la méthode de Gauss

Le coût est évalué en nombre d'opérations élémentaires nécessaires pour le calcul de la solution.

1. Le calcul des coefficients $a_{ij}^{(k)}$ $(k = 1, 2, \cdots, n;$ $i, j = k+1, k+2, \cdots, n)$ par la formule () nécessite :
   - $\sum_{k=1}^{n-1} (n-k) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$ divisions;
   - $\sum_{k=1}^{n-1} (n-k)^2 = \frac{n(n-1)(2n-1)}{6}$ multiplications.
   - $\sum_{k=1}^{n-1} (n-k)^2 = \frac{n(n-1)(2n-1)}{6}$ additions.
2. Le calcul des $b_i^{(k)}$ nécessite
   - n divisions;
   - $\sum_{k=1}^{n-1} (n-k) = \frac{n(n-1)}{2}$ multiplications.
   - $\sum_{k=1}^{n-1} (n-k) = \frac{n(n-1)}{2}$ additions.

Au bilan, la phase d'élimination nécessite de l'ordre de $\frac{2n^3}{3}$ opérations arithmétiques élémentaires. La phase de substitution nécessite n^2 opérations arithmétiques élémentaires.

Méthodes de Choleski (Factorisation LL^t)

La méthode de CHOLESKI pour la résolution du système (15) consiste à la factorisation de la matrice A, lorsqu'elle est ***symétrique et définie positive***, sous la forme :

$$A = LL^t \tag{19}$$

où L est une matrice triangulaire inférieure inversible (construite par identification), et L^t la matrice transposée de L.
Ainsi, le système (15) s'écrit :

$$LL^t x = b. \tag{20}$$

Si on pose :

$$L^t x = y, \tag{21}$$

on résout d'abord :

$$Ly = b \tag{22}$$

par la méthode de la descente (puisque L est triangulaire inférieure) et ensuite

$$L^t x = y, \tag{23}$$

par la méthode de la remontée (la matrice étant triangulaire supérieure).
La méthode de Choleski procède en trois étapes :

1^{ere} **étape** : *Calcul des coefficients de la matrice* L

$$\left.\begin{array}{l} L_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \\ L_{ji} = \dfrac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right), \quad j = i+1, ..., n \end{array}\right\}, \quad i = 1, ..., n \tag{24}$$

2^{eme} **étape**: R*ésolution du système triangulaire inférieur* $Ly = b$

$$\begin{array}{l} y_1 = \dfrac{b_1}{L_{11}} \\ y_i = \dfrac{\left(b_i - \sum_{k=1}^{i-1} L_{ik} y_k \right)}{L_{ii}}, \quad i = 2, ..., n \end{array} \tag{25}$$

3^{eme} **étape**: *Résolution du système triangulaire supérieure* $L^t x = y$

$$
\begin{aligned}
x_n &= \frac{y_n}{L_{nn}} \\
x_i &= \frac{\left(y_i - \sum_{j=i+1}^{n} L_{ji} x_j \right)}{L_{ii}}, \quad i = n-1, ..., 1
\end{aligned}
\tag{26}
$$

Nombre d'opérations
La méthode de CHOLESKI requiert environ $\frac{n^3}{6}$ additions, $\frac{n^3}{6}$ multiplications et n extractions de racines carrées ; il y a donc avantage à utiliser Choleski par rapport à Gauss lorsque la matrice des coefficients du système est symétrique et définie positive, de plus cette décomposition est stable.

Outline

Outline

Outline