Cours 1: Introduction

- Présentation du cours
- Généralités
- Un exemple introductif

Programmation 1. L.1.2

But : donner une connaissance de base du langage C à tous les étudiants de sciences.

Organisation: douze semaines d'enseignement découpées en :

- Cours magistral (1,5 heure).
- Travaux dirigés (2 heures).
- Travaux pratiques (2 heures)

Manuels recommandés :
Concepts Fondamentaux de l'Informatique, A. Aho, V. Ullman, Dunod, 1993.
Méthodologie de la programmation en C, Jean-Pierre Braquelaire, Masson, 1994.
Le langage C, Brian Kernighan et Dennis Ritchie, Masson, 1988.
Passeport pour Unix et C, Jean-Marc Champarnaud et Georges Hansel, Eyrolles, 2000.

Contrôle des connaissances

1. Contrôles réalisés en TD et TP.
2. Projet réalisé par binômes (sujets remis en milieu de semestre et projet présenté durant le dernier TP). Un projet réalisé en binômes par deux étudiants de groupes de TP différents est à rendre à un même chargé de TP.
3. Examen en fin de semestre (examen écrit avec documents).

Un exemple étonnant

Que fait le programme suivant ?

```
long a = 10000, b, c = 8400, d, e, f[8401], g;

int main(void) {
for (; b-c;) f[b++] = a/5;
for (; d = 0,g = c * 2;c -= 14,printf("%.4ld", e+d/a),e =d % a)
for (b = c; d += f[b] * a, f[b] = d%--g, d /= g--, --b; d *= b);
return 0;
}
```

Réponse

3141592653589793238462643383279502884197169399375105820974944592
2148086513282306647093844609550582231725359408128481117450284102
8810975665933446128475648233786783165271201909145648566923460348
7006606315588174881520920962829254091715364367892590360011330530
0365759591953092186117381932611793105118548074462379962749567351
2440656643086021394946395224737190702179860943702770539217176293
5263560827785771342757789609173637178721468440901224953430146549
9021960864034418159813629774771309960518707211349999998372978049
8302642522308253344685035261931188171010003137838752886587533208
8731159562863882353787593751957781857780532171226806613001927876
3278865936153381827968230301952035301852968995773622599413891249
0829533116861727855889075098381754637464939319255060400927701671
0181942955596198946767837449448255379774726847104047534646208046
2056966024058038150193511253382430035587640247496473263914199272
9924586315030286182974555706749838505494588586926995690927210797
5499119881834797753566369807426542527862551818417574672890977772
2350141441973568548161361157352552133475741849468438523323907394

92221842725502542568876717904946016534668049886272327917860857840064225125205117392984896084128488626945604241965285022210661186636437191728746776465757396241389086583264599581339047802759009905224894077267194782684826014769909026401363944374553050682034961696461515709858387410597885959772975498930161753928468138268683252468084598727364469584865383673622262609912460805124388439045 1

Un exemple introductif

On veut calculer par un programme en C une table de conversion de Francs en Euro comme

```
10    1
20    3
30    4
40    6
50    7
60    9
70    10
80    12
90    13
100   15
```

On rappelle qu'un euro vaut 6,50 francs. La table ci-dessus est arrondie à l'euro inférieur.

Le programme

```
#include <stdio.h>

int main(void) {

/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/

  int euro, francs;
  int bas, haut, pas;

  bas = 10;
  haut = 100;
  pas = 10;
  francs = bas;
  while (francs <= haut) {
    euro = francs / 6.50;
    printf("%d\t%d\n",francs,euro);
    francs = francs + pas;
  }
  return 0;
}
```

Analyse

Les deux lignes

```
/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/
```

sont un *commentaire*. Ensuite

```
int euro, francs;
int bas, haut, pas;
```

sont des *déclarations*. Les variables sont déclarées de type entier (`int`). On a d'autres types en C comme `float` pour les nombres réels.

Les lignes suivantes

```
bas = 10;
haut = 100;
pas = 10;
francs = bas;
```

sont des *instructions d'affectation*. Une instruction se termine par ;

La boucle d'itération

Chaque ligne de la table se calcule de la même façon. On utilise donc une *itération* sous la forme:

```
while (francs <= haut) {
   ...
}
```

Signification : on teste d'abord la condition. Si elle est vraie, on execute le corps de la boucle (les instructions entre accolades). Ensuite on teste de nouveau la condition. Si elle est vraie on recommence, et ainsi de suite.

Si elle est fausse, on passe à la suite (ici à la fin).

La fonction printf

L'instruction

```
printf("%d\t%d\n",francs,euro);
```

est un appel de la fonction `printf`. Son premier argument donne le format d'affichage: Les `%d` sont des indications pour l'affichage des arguments suivants. Le `d` est là pour indiquer une valeur entière (en décimal). Si on avait écrit

```
printf("%3d %6d\n",francs, euro);
```

on aurait écrit sur un nombre fixe de caractères (3 et six) d'où le résultat justifié à droite :

```
 10      1
 20      3
 ...
 70     10
 80     12
 90     13
100     15
```

Et les centimes?

```
#include <stdio.h>

int main(void) {

/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/

  float euro, francs, taux;
  int bas, haut, pas;

  bas = 10;
  haut = 100;
  pas = 10;
  francs = bas; taux = 1/6.5;
  while (francs <= haut) {
    euro = francs * taux;
    printf("%3.0f %6.2f\n",francs,euro);
    francs = francs + pas;
  }
 return 0;
}
```

Résultat

```
10     1.54
20     3.08
30     4.62
...
```

Si on avait utilisé `taux = 1/6.5` avec taux déclaré entier, on aurait obtenu 0 (facheux).

Le format `%6.2f` signifie : nombre réel écrit sur six caractères avec deux décimales. Pour `%3.0`, c'est sans décimale ni point.

L'instruction for

Autre forme du programme :

```
#include <stdio.h>

int main(void) {
  int fr;

  for (fr = 10; fr <= 100; fr = fr + 10)
    printf("%3d %6.2f\n", fr, fr/6.5);
  return 0;
}
```

L'initialisation

```
fr = 10
```

est faite d'abord. On teste ensuite la condition

```
fr <= 100
```

Si elle est vraie, on exécute le corps de la boucle.
On exécute ensuite l'instruction d'incrémentation

```
fr = fr + 10;
```

On reteste la condition et ainsi de suite.

Constantes symboliques

On peut utiliser des définitions préliminaires pour définir des constantes.

```
#include <stdio.h>

#define BAS 10
#define HAUT 100
#define PAS 10

int main(void) {
  int fr;

  for (fr = BAS; fr <= HAUT; fr = fr + PAS)
    printf("%3d %6.2f\n", fr, fr/6.5);
  return 0;
}
```

Cours 2 : Le langage C

Le langage C a été crée en 1970 aux Bell Laboratories par Brian Kernighan et Denis Ritchie.

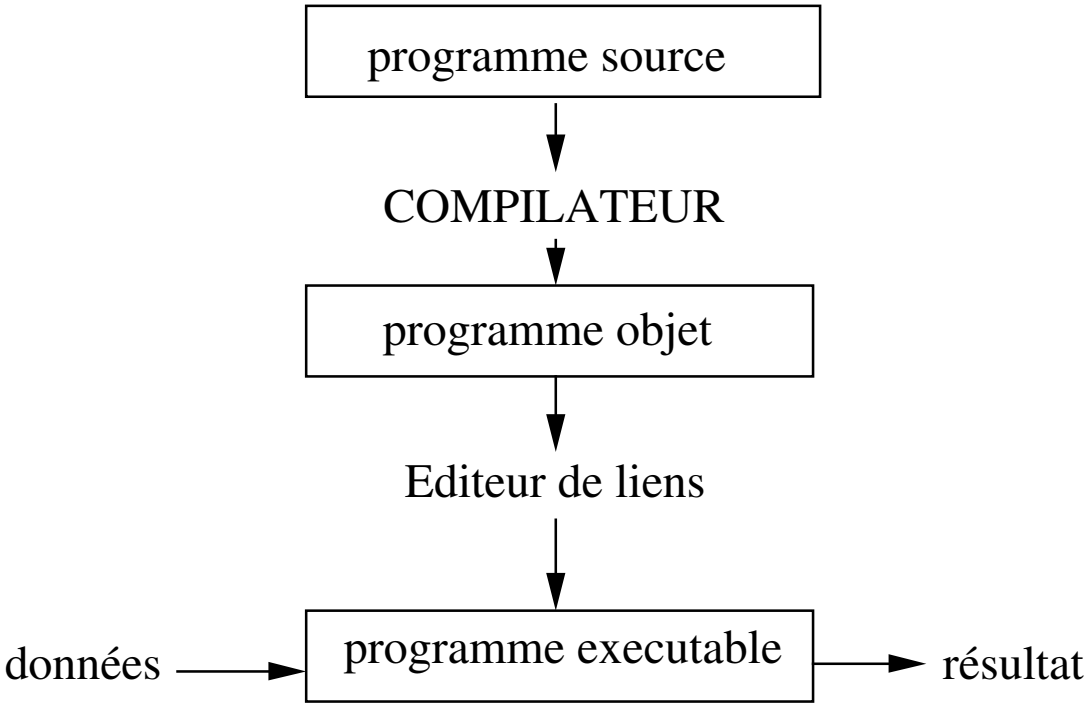
Il existe beaucoup d'autres langages de programmation.

- Fortran (surtout pour le calcul numérique).
- Cobol (pour la gestion).
- Pascal (ancien).
- Ada (pour les grandes applications)
- Java,...

Ce sont des langages de *haut niveau* par opposition au langage machine et à l'assembleur, dits de *bas niveau*.

Il existe par ailleurs un très grand nombre de langages *spécialisés* allant des langages de commande des imprimantes aux systèmes de calcul formel.

Exécution d'un programme



Exemple de programme en C

Affiche au terminal la chaîne de caractères *bonjour* et passe à la ligne.

```
------------------------------
#include <stdio.h>

int main(void) {
  printf("bonjour\n");
  return 0;
}
------------------------------
```

ou encore

```
#include <stdio.h>

void afficher(void){
 printf("bonjour\n");
}
int main(void) {
 afficher();
 return 0;
}
```

Deuxième exemple

Programme qui lit un entier et écrit son carré.

```
int main(void){
 int n,m;
 printf("donnez un nombre :");
 scanf("%d",&n);
 m=n*n;
 printf("voici son carre :");
 printf("%d\n",m);
 return 0;
}
```

Execution:

```
donnez un nombre : 13
voici son carre : 169
```

Structure d'un programme C

Un programme C est constitué de fonctions.
Elles ont toutes la même structure.

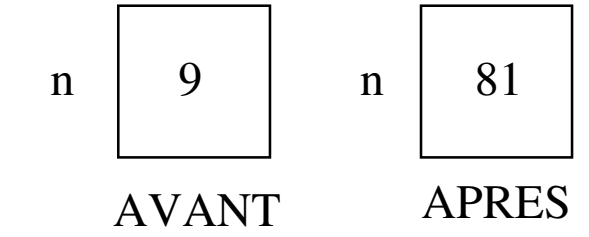
Type retourné Nom(*Types et Noms des paramètres*)
{
déclarations
instructions
}

La fonction principale s'appelle `main`.

Les variables

Une *variable* en C est désignée par un nom qui est une chaîne de caractères (commençant par une lettre) appelée un *identificateur*.

Une variable a une *adresse*, un *type* et une *valeur*.



La mémoire centrale

Les données sont rangées dans la mémoire sous forme de suites de *bits* égaux à 0 ou 1.

Les bits sont regroupés par groupe de 8 (un octet) puis, selon les machines, par *mots* de 16, 32 ou 64 bits.

Chaque mot porte un numéro: son *adresse* en mémoire.

On utilise les octets pour coder les caractères en utilisant le *code ASCII*. On aura par exemple pour la lettre A le code:

place	7	6	5	4	3	2	1	0
valeur	0	1	0	0	0	0	0	1

qui représente 65 en binaire.

Le premier bit est utilisé comme bit de parité en général (non standard). Sur les 7 autres, on peut coder $2^7 = 128$ caractères (pas beaucoup pour les accents!).

Les types

Chaque variable a un *type*. Elle est *déclarée* avec son type par:

```
int n;
```

Les types de base en C sont:

`char`	caractères
`int`	entiers
`float`	flottants
`double`	flottant double précision

Il existe aussi des types plus compliqués : les *tableaux* et les *structures*.

Les entiers

Le type entier (`int`) est représenté sur 16 ou 32 bits selon les machines (à tester dans `<limits.h>`).

Sur 16 bits, les valeurs vont de $-2^{15} = -32768$ à $2^{15} - 1 = 32767$.

Sur 32 bits, on a comme valeurs les entiers relatifs de -2^{31} à $2^{31} - 1$.

Les opérations sont $+$, $-$, $*$, $/$ (division entière par suppression de la partie fractionnaire) et $\%$ (reste de la division euclidienne).

Les réels

C'est le type `float` avec une représentation en *flottant* de la forme 1.25×10^{-4}. Ils sont représentés en général sur 32 bits avec six chiffres significatifs et une taille comprise entre 10^{-38} et 10^{+38}.

Les opérateurs sont $+, -, *, /$, ...

Il y a des *conversions* entre les divers types. Par exemple, si x est de type `float`, x+1 est de type float.

Il y aussi un type `double` pour les réels en double précision.

Les caractères

Le type `char` est représenté sur un seul octet. Ce sont les caractères représentés dans le code ASCII. C'est en fait un entier.

Les constantes de type `char` s'écrivent `'x'`.

Attention `'0'` vaut 79, pas 0. C'est `'\0'` qui vaut zéro.

Certains caractères s'écrivent de façon particulière, comme `'\n'` (newline).

On forme avec des caractères des chaînes comme

```
"bonjour\n"
```

Les expressions

On construit une expression en utilisant des variables et des constantes puis des opérateurs choisis parmi

- les opérateurs arithmétiques : $+, -, *, /, ...$
- Les comparateurs : $>, >=, ==, !=, \ldots$
- Les opérateurs logiques : && (et), || (ou)

Elle peut être de type char, int ou float.
Exemple: si i, j sont entiers et x réel alors:
$i - j$ est entier
$2 * x$ est réel
'2'<'A' est entier (la valeur 'vrai' est représentée par l'entier 1)

Expressions logiques

Les valeurs logiques 'vrai' et 'faux' sont représentées en C par les entiers 1 et 0 (en fait toute valeur non nulle est interprétée comme vrai).

Les opérateurs logiques sont `&&` (et), `||` (ou) et `!` (non) avec les *tables de vérité* :

ou	0	1
0	0	1
1	1	1

et	0	1
0	0	0
1	0	1

Le C pratique l'évaluation *paresseuse* : si C est vraie, alors `C || D` est vraie, même si D n'est pas définie.
De même, si C est fausse, `C && D` est fausse.

Exemple

Conversion des températures des Fahrenheit aux Celsius. La formule est

$$C = \frac{5}{9}(F - 32)$$

```
int main(void)
{
 float F, C;

 printf("Donnez une temperature
 en  Fahrenheit : ");
 scanf("%f",&F);
 C = (F - 32) * 5 / 9;
 printf("Valeur en  Celsius : %3.1f\n",C);
 return 0;
}
```

Exécution :

donnez une temperature en Fahrenheit : 100 valeur en Celsius : 37.8

Les déclarations

On peut regrouper des déclarations et des instructions en un bloc de la forme

{
liste de déclarations
liste d'instructions
}

Les déclarations sont de la forme *type nom ;* comme dans

```
int x;
```

ou *type nom=valeur ;* comme dans

```
int x=0;
```

qui initialise x à la valeur 0.

L'affectation

C'est le mécanisme de base de la programmation. Elle permet de changer la *valeur* des variables.

```
x = e;
```

affecte à la variable `x` la valeur de l'expression `e`. Attention : A gauche de =, on doit pouvoir calculer une adresse (on dit que l'expression est une *l-valeur*).

Exemples

```
i = i+1;
```

augmente de 1 la valeur de i. Aussi réalisé par l'instruction `i++;`

```
temp = i; i = j; j = temp;
```

échange les valeurs de i et j.

Instructions conditionnelles

Elles ont la forme:

if (*test*)
  instruction

(forme incomplète)

ou aussi:

if (*test*)
  instruction
else
  instruction

(forme complète)

Le test est une expression de type entier comme `a<b` construite en général avec les comparateurs `<`, `==`, `!=`, ... et les opérateurs logiques `&&`, `||`, ...

Exemple

Calcul du minimum de deux valeurs a et b :

```
if (a < b){
  min = a;
} else {
  min = b;
}
```

Forme abrégée :

```
min = (a<b)? a : b;
```

Minimum de trois valeurs :

```
if (a < b && a < c)
    min =  a;
  else if (b < c)
    min = b;
  else
    min = c;
```

Branchements plus riches

Si on a un choix avec plusieurs cas, on peut utiliser une instruction d'aiguillage.

Par exemple, pour écrire un chiffre en lettres:

```
switch (x){
  case 0: printf("zero"); break;
  case 1: printf("un"); break;
  case 2: printf("deux"); break;
  ...
  case 9: printf("neuf"); break;
  default: printf("erreur");
}
```

Cours 3 : Les itérations

1. Les trois formes possibles
2. Traduction
3. Itération et récurrence
4. Ecriture en binaire

Les trois formes possibles

Elles ont l'une des trois formes suivantes:

1. boucle **pour**.
2. boucle **tant que** faire.
3. boucle faire **tant que**.

Boucle pour

`for` (*initialisation*; *test*; *incrémentation*) {
  liste d'instructions
}

comme dans:

```
for (i = 1;  i<=n; i++) {
  x = x+1;
}
```

L'exécution consiste à itérer
initialisation
(test, liste d'instructions, incrémentation)
(test, liste d'instructions, incrémentation)
....
On peut aussi mettre une liste d'instructions à la place de incrémentation.

Boucle tant que

`while` (*test*) {
  liste d'instructions
}
  comme dans:

```
while (i <= n) {
 i = i+1;
}
```

Si $i \leq n$ avant l'exécution, on a après $i = n+1$.
Sinon, i garde sa valeur.

L'autre boucle tant que

C'est la boucle 'faire tant que'.

`do` {
  liste d'instructions
} `while` (*test*);
  comme dans:

```
do {
 i = i+1;
} while (i <= n);
```

Si $i \leq n$ avant, on a après $i = n + 1$. Sinon, i augmente de 1.

Exemples de boucle pour

Calcul de la somme des n premiers entiers:

```
s = 0;
for (i = 1; i<= n; i++) s = s+i;
```

Après l'exécution de cette instruction, `s` vaut:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Calcul de la somme des n premiers carrés:

```
s = 0;
for (i = 1; i<= n; i++) s = s+i*i;
```

Après l'execution de cette instruction, `s` vaut:

$$1 + 4 + 9 + \ldots + n^2$$

(au fait, ça fait combien?)

Exemple de boucle tant que

Calcul de la première puissance de 2 excédant un nombre N:

```
p = 1;
while (p < N) {
  p = 2*p;
}
```

Résultats:

Pour N=100 on a p=128

Pour N=200 on a p=256

...

Test de primalité

Voici un exemple de boucle 'do while': le test de primalité d'un nombre entier.

```
int main(void){
 int d, n, r;
  printf("donnez un entier: ");
  scanf("%d", &n);
  d = 1;
  do {
   d = d+1;
   r = n % d;
  } while (r >= 1 &&  d*d <= n);
  if (r==0)
        printf("nombre divisible par %d\n",d);
   else
        printf("nombre premier\n");
  return 0;
}
```

Traduction des itérations

L'instruction

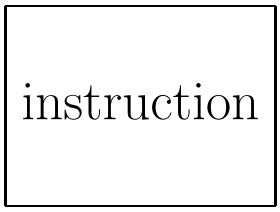
`while` (i <= n) {

instruction

}

se traduit en assembleur par:

```
1: IF i > n  GOTO 2
```



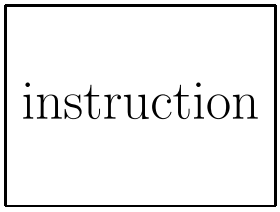

```
   GOTO 1
2:
```

De même l'itération

```
for (i = 1; i<= n; i++) {
  instruction
}
```

est traduite par:

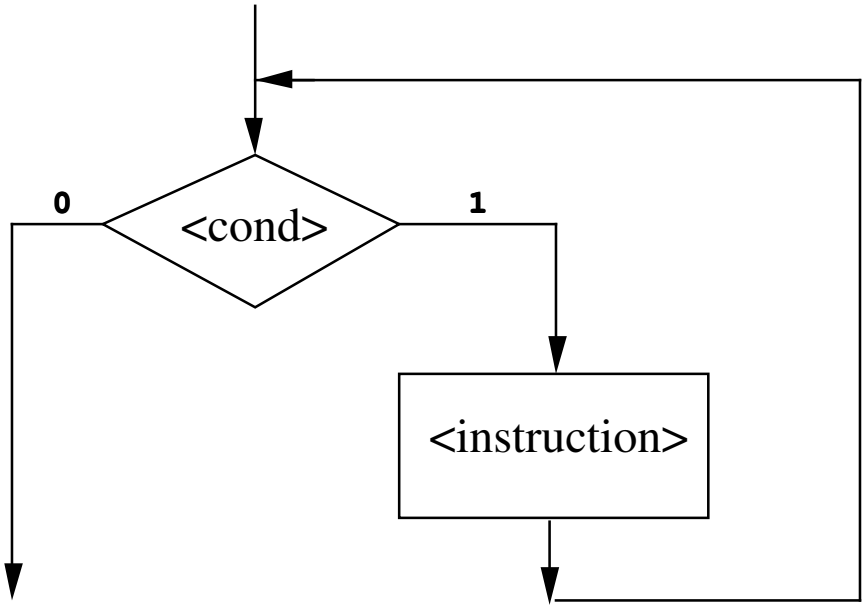
```
   i = 1
1: if i > n GOTO 2
```




```
   i = i+1
   GOTO 1
2:
```

Organigrammes

On peut aussi traduire en schémas appelés *organigrammes* ou *chartes*.



Temps de calcul

Les itérations permettent d'écrire des programmes qui tournent longtemps. Parfois trop!
Le programme

```
int main(){
 while (1);
 return 0;
}
```

ne s'arrête jamais.

Itération et récurrence

Les programmes itératifs sont liés à la notion mathématique de *récurrence*.

Par exemple la suite de nombre définie par récurrence par $f_0 = f_1 = 1$ et pour $n \geq 2$ par

$$f_n = f_{n-1} + f_{n-2}$$

s'appelle la suite de Fibonacci:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Elle peut être calculée par le programme suivant (qui affiche les 20 premières valeurs):

```
int main(void){
 int i,u,v,w;

  u = 1; v = 1;
  for (i= 1; i<= 20; i++) {
     w = u+v;
     printf("%d ",w);
     u = v; v = w;
  }
 return 0;
}
```

Résultat :

```
   2    3    5    8   13   21   34
  55   89  144  233  377  610  987
1597 2584 4181 6765 10946 17711
```

Invariants de boucles

Pour *raisonner* sur un programme comportant des itérations, on raisonne par récurrence.

On utilise une propriété vraie à chaque passage dans la boucle, appelée *invariant de boucle*. Par exemple, dans le programme précédent, la propriété:

$$P(i) : \ u = f_{i-1}, v = f_i$$

est un invariant de boucle: elle est vraie à chaque fois qu'on entre dans la boucle.

Pour le démontrer, on doit

1. vérifier qu'elle est vraie à la première entrée (et donc pour $i = 1$).
2. Que si elle est vraie à un passage (pour i), elle est vraie au suivant (pour $i + 1$).

Un autre exemple : La fonction factorielle

Le programme suivant

```
int main(void) {

 int i, n, fact;

  scanf("%d",&n);
  fact=1;
  for (i=2; i<= n; i++)
   fact = fact * i;
  printf("n!=%d\n",fact);
  return 0;
}
```

calcule

$$n! = 1 \times 2 \times \cdots \times n$$

L'invariant de boucle est $fact = (i-1)!$.

Un exemple plus difficile : l'écriture en binaire

La représentation binaire d'un nombre entier x est la suite $(b_k, \ldots, b_1, b_0)$ définie par la formule:

$$x = b_k 2^k + \cdots + b_1 2 + b_0$$

Principe de calcul : en deux étapes

1. On calcule $y = 2^k$ comme la plus grande puissance de 2 t.q. $y \leq x$.
2. Itérativement, on remplace y par $y/2$ en soustrayant y de x à chaque fois que $y \leq x$ (et en écrivant 1 ou 0 suivant le cas).

```
int main(void) {

int x,y,n;

  scanf("%d", &n); x = n; y = 1;
  while (y+y <= x)  y = y+y;
  while (y != 0) {
    if (x < y) printf("0");
    else {
      printf("1"); x = x - y;
    }
    y = y / 2;
  }
  printf("\n");
  return 0;
}
```

Cours 4
Fonctions

Idée : écrire un programme comme un jeu de fonctions qui s'appellent mutuellement (informatique en kit).

Chaque fonction aura:

1. Une *définition*: qui comprend son nom et
   (a) Le type et le nom de ses paramètres.
   (b) Le type de la valeur rendue.
   (c) Son code : comment on la calcule.
2. Un ou plusieurs *appels*: c'est l'utilisation de la fonction.
3. Un ou plusieurs *paramètres*: ce sont les arguments de la fonction.
4. Un *type* et une *valeur*.

Exemples sans paramètres

Nous avons déja vu la fonction

```
void afficher(void){
 printf("bonjour\n");
}
```

Le type `void` est utilisé par convention en l'absence de valeur ou de paramètres. On peut aussi simplement écrire

```
double pi(void){
 return 3.1415926535897931;
}
```

dont la valeur est de type `double` (nombres réels en double précision). On pourra ensuite, dans une autre fonction, écrire l'expression `0.5*pi()`.

Exemple avec paramètres

La fonction suivante rend le maximum de deux valeurs de type flottant.

```
float fmax(float a, float b){
 if (a > b)
   return a;
 else
   return b;
}
```

de sorte que l'expression `fmax(pi()*pi(),10.0)` vaut 10.

Version plus courte

```
float fmax(float a, float b){
 return (a > b) ? a : b;
}
```

Définition de fonctions

La définition d'une fonction comprend deux partie: l'en-tête (ou prototype) qui est de la forme

Type de valeur Nom (*Types des paramètres*)

suivi de son code : un bloc de la forme

{
liste-de déclarations
liste-d'instructions
}

Les déclarations sont valables à l'intérieur du bloc seulement (variables locales).

On peut aussi avoir des variables *globales* déclarées au même niveau que les fonctions.

L'instruction `return` *expression*; permet de rendre une valeur et arrête l'exécution.

Une fonction doit être définie avant d'être utilisée.

Appels de fonctions

Une fonction est déclarée avec des paramètres formels. Elle est appelée avec des paramètres réels (en nombre égal).

déclaration de fonction	appel de fonction
paramètre formel	paramètre d'appel

```
float fmax(float a, float b){
 return (a> b) ? a : b;
}

int main(void){
 float x;
 scanf("%f",&x);
 printf("%f",fmax(x,x*x));
 return 0;
}
```

Autres exemples de fonctions

Pour tester si un nombre est premier:

```
void premier(int n){
 int d;

  d = 1;
  do
   d = d+1;
  while (n % d >= 1 &&  d*d <= n);
  if (n % d == 0)
        printf("nombre divisible par %d\n",d);
   else
        printf("nombre premier\n");
}
```

La fonction factorielle:

```
int fact(int n){
 int i, f;

  f = 1;
  for (i=2; i<= n; i++)
   f = f * i;
  return f;
}
```

Modes de transmission des paramètres

L'appel d'une fonction

```
truc(x)
```

transmet à la fonction `truc` la *valeur* de la variable x (ou de l'expression x).

Pour changer la valeur de la variable x, il faut transmettre son *adresse*. On utilise l'opérateur & comme dans

```
scanf("%d",&n);
```

qui permet de lire au clavier une valeur qui sera affectée à la variable n.

Exemple

```
void echange (int x, int y){
int t;

t = x; x = y; y = t;
}
```

L'appel de `echange(a,b)` ne change pas les valeurs de a et b: la fonction `echange` travaille sur des copies auxquelles on passe la *valeur* de a et b.

Par contre:

```
void echange (int *p, int *q){
int temp;

temp = *p; *p = *q; *q = temp;
}
```

Réalise l'échange en appelant `echange(&a,&b);`

Explication

Les opérateurs `&` (*référence*) et `*` (*déréférence*) sont inverses l'un de l'autre.

- Si x est une variable de type t, alors `&x` est l'adresse de x. C'est une expression de type $t*$
- Si p est déclaré de type $t*$, alors `*p` est une expression de type t.

On dit que p est un *pointeur*. C'est une variable dont la valeur est une adresse.

Exemple

Supposons que l'adresse de la variable x soit 342 et sa valeur 7.

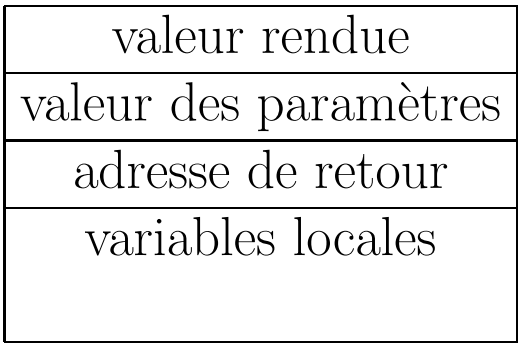
L'instruction

```
p=&x;
```

donne à la variable p la valeur 342. La valeur de l'expression `*p` est 7.

Implémentation des appels de fonctions

Chaque appel de de fonction est effectué dans une zone mémoire nouvelle et réservée à cet appel.



On dit que cette zone est un *enregistrement d'activation* (‘*activation record*’). Quand l'appel de fonction est terminé, la zone est libérée (on dit que c'est une gestion *dynamique* de la mémoire).

Cours 5
Tableaux

Type de données *construit* le plus simple. Permet de représenter des suites d'éléments d'un ensemble.

0	1	2	3	4
0.3	1.4	15.0	1.6	4.8

Avantage: pouvoir accéder aux éléments avec une adresse calculée.
Contrainte : tous les éléments doivent être du même type.

Déclaration :

```
float a[5];
```

Tous les indices commencent à 0. Le *nom* de l'élément d'indice i est:

```
a[i]
```

Définition des tableaux

On utilise souvent une *constante* pour désigner la dimension d'un tableau.

```
#define N 50 /* dimension des tableaux */
..
float Resultats[N];
```

déclare un tableau de 50 nombres réels de R[0] à R[49].

On peut définir et initialiser un tableau de la façon suivante.

```
int a[5]={1,1,1,1,1};
```

initialise tous les éléments de a à 1.

Lecture et écriture d'un tableau

On peut lire les valeurs d'un tableau de N nombres entiers

```
void Lire(int a[]){
 int i;
 for (i = 0; i < N; i++)
   scanf("%d",&a[i]);
}
```

ou les écrire

```
void Ecrire(int a[]){
 int i;
 for (i = 0; i < N; i++)
   printf("%d ",a[i]);
 printf(``\n'');
}
```

Le passage est par adresse car le nom d'un tableau est une adresse (celle du premier élément). De ce fait on peut écrire

```
void Lire(int * a)
```

au lieu de `void Lire(int a[])`.

Si on veut passer la dimension du tableau en paramètre (au lieu d'utiliser une constante symbolique), il faut écrire :

```
void Lire(int a[], int n){
 int i;
 for (i = 0; i < n; i++)
   scanf("%d",&a[i]);
}
```

et

```
void Ecrire(int a[], int n){
 int i;
 for (i = 0; i < n; i++)
   printf("%d ",a[i]);
 printf(``\n'');
}
```

On pourra alors écrire par exemple

```
int main(void){
  int a[N];
  scanf("%d",&n);
  if(n>N) printf(``erreur\n'');
```

```
  Lire(a,n);
  Ecrire(a,n);
  return 0;
}
```

Recherche du minimum

Donnée: Tableau $a[0], a[1], \ldots, a[N-1]$ de nombres entiers.
Principe: On garde dans la variable m la valeur provisoire du minimum et on balaye le tableau de 0 à $N-1$.

```
int Min(int a[]){
 int i, m;

  m = a[0];
  for  (i = 1; i < N; i++)
   if (a[i] < m)  m = a[i];
  return m;
}
```

Somme des éléments

Calcul de la somme $a[0] + a[1] + \ldots + a[N-1]$.

```
int Somme(int a[]){
  int i,s;
  s=0;
  for(i=0;i<N;i++)
    s+=a[i];
  return s;
}
```

Tri par sélection

Donnée: N nombres entiers $a[0], a[1], \ldots, a[N-1]$.

Résultat: Réarrangement croissant $b[0], b[1], \ldots, b[N-1]$ des nombres $a[i]$.
Exemple:

$$a = \begin{array}{|c|c|c|c|c|}\hline 24 & 5 & 81 & 2 & 45 \\ \hline\end{array}$$
$$b = \begin{array}{|c|c|c|c|c|}\hline 2 & 5 & 24 & 45 & 81 \\ \hline\end{array}$$

Principe: A la i-ème étape, on cherche le minimum de $a[i], a[i+1], \ldots, a[N-1]$ et on l'échange avec $a[i]$.

Tri par sélection

```
#define  N 10
/* prototypes*/
void Lire(int a[]);
void Ecrire(int a[]);

int IndiceMin(int a[], int i){
/* calcule l'indice du minimum*/
/* a partir de i */
  int j,k;

  j = i;
  for  (k = i+1; k < N; k++)
    if (a[k] < a[j])  j = k;
  return j;
}
```

```
void Trier(int a[]){
  int i;

  for (i = 0; i < N; i++){
     int j,t;
     j=IndiceMin(a,i);
     t=a[j]; a[j]=a[i]; a[i]=t;
  }
}

int main(void){
  int a[N];
  Lire(a); Trier(a); Ecrire(a);
  return 0;
}
```

Le nombre d'opérations est de l'ordre de N^2. Pour $N = 10^5$ avec 10^6 opérations/s, le temps devient de l'ordre de 10^4 s (près de 3 h).

Interclassement

On veut réaliser la fusion de deux suites croissantes $a[0] < a[1] < \cdots < a[n-1]$ et $b[0] < b[1] < \cdots < b[m-1]$, c'est à dire une suite $c[0] < c[1] < \cdots < c[m+n-1]$ qui est un interclassement des suites a et b.

Le principe est de parcourir linéairement les suites a et b de gauche à droite en insérant à chaque fois dans c le plus petit des deux éléments courants.

Par exemple: Si

$$a = \boxed{2}\boxed{10}\boxed{12}\boxed{25}\boxed{41}\boxed{98}$$
$$b = \boxed{5}\boxed{14}\boxed{31}\boxed{49}\boxed{81}$$

Alors:

$$c = \boxed{2}\boxed{5}\boxed{10}\boxed{12}\boxed{14}\boxed{25}\boxed{31}\boxed{41}\boxed{49}\boxed{81}\boxed{98}$$

On utilise deux éléments supplémentaires à la fin des tableaux a et b plus grands que tous les autres (sentinelles).

```
void Interclassement(int a[], int b[], int c[],
  /*i et j servent a parcourir a et b*/
  int i = 0, j = 0;
  while (i+j < p)
    if ( a[i] <= b[j] ){
      c[i+j] = a[i];
      i++;
    }
    else if (b[j]< a[i]){
      c[i+j] = b[j];
      j++;
    }
}
```

La fonction principale s'écrit alors

```
int main(void){
  int a[N]; int b[M];
  int k=N+M-2; int c[k];
  Lire(a,N); Lire(b,M);
```

```
  Interclassement(a,b,c,k);
  Ecrire(c,k);
  return 0;
}
```

Inversion d'une table

Problème: On dispose d'une table qui donne pour chaque étudiant(e) (classé dans l'ordre alphabétique) son rang au concours de (beauté, chant, informatique,..):

numéro	nom	rang
0	arthur	245
1	béatrice	5
2	claude	458
	...	

On veut construire le tableau qui donne pour chaque rang le numéro de l'étudiant:

rang	nom	numéro
0	justine	125
1	nathan	259
	...	

Programmation

On part du tableau `int rang[N]` et on calcule le résultat dans un tableau `numero` du même type.

```
void Inversion(int rang[], int numero[])
{
 int i;
 for (i = 0; i < N; i++)
     numero[rang[i]] = i;
}
```

On a en fait réalisé un *tri* suivant une clé (le rang au concours) par la méthode de *sélection de place*.

Moralité : pour parcourir un tableau le plus efficace n'est pas toujours de le faire dans l'ordre. Ici on utilise un *adressage indirect*.

Tri par séléction de place

On fait l'hypothèse que toutes les valeurs sont $<$ M. On utilise un tableau de M booléens.

```
void Tri(int a[]){
  int i,j;
  int aux[M];
  for(i=0;i<M; i++) aux[i]=0;
  for(i=0;i<N; i++) aux[a[i]]=1;
  j=0;
  for(i=0;i<M; i++)
    if(aux[i]==1){
      a[j]=i;
      j++;
    }
}
```

Résultat : le nombre d'opérations est de l'ordre de M.

Cours 6
Tableaux (suite)

- tableaux à deux dimensions
- débordements
- chaînes de caractères

Tableaux à deux dimensions

Un tableau à deux dimensions permet d'accéder à des informations comme une image (en donnant pour chaque point (i, j) une valeur comme la couleur).

L'élément en ligne i et colonne j est noté

```
a[i][j]
```

Du point de vue mathématique, c'est une *matrice*. Voici par exemple comment on initialise un tableau $N \times N$ à 0.

```
int C[N][N];
int i, j;

for (i = 0; i < N, i++)
 for (j = 0; j < N; j++)
     C[i][j] = 0;
```

Les tableaux à plusieurs dimensions sont rangés en mémoire comme un tableau à une seule dimension. Si on range (comme en C) les éléments par ligne alors $a[i][j]$ est rangé en place $i \times m + j$ (pour un tableau $[0..n-1] \times [0..m-1]$).
Pour passer un tableau à deux dimensions en paramètre, on doit donner la deuxième dimension. Exemple : lecture d'un tableau à deux dimensions.

```
void Lire2(int a[][N]){
  int i,j;
  for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    scanf("%d",&a[i][j]);
}
```

L'expression `a[i]` désigne la ligne d'indice i du tableau. On peut donc aussi écrire :

```
void Lire2(int a[][N]){
  int i;
```

```
  for(i=0;i<N;i++)
    Lire(a[i]);
}
```

Exemple

Calcul des sommes par lignes d'un tableau.

```
void Somme2(int a[][N],int s[]){
  int i;
  for(i=0;i<N;i++)
    s[i]=Somme(a[i]);
}

int main(void){
  int a[N][N];
  int s[N];
  Lire2(a);
  Somme2(a,s);
  Ecrire(s);
  return 0;
}
```

Débordements

Le langage C ne réalise pas de contrôle sur les valeurs des indices. On doit faire attention à ne jamais appeler un indice hors des valeurs autorisées.

Sinon, il peut se produire des choses désagréables:

1. modifications de valeurs d'autres variables.
2. erreur d'execution

Exemples

L'affectation change la valeur d'une autre variable:

```
int main(void){
  int x=0;
  int a[5];
  a[5]=1;
  printf("%d\n",x);
  return 0;
}
```

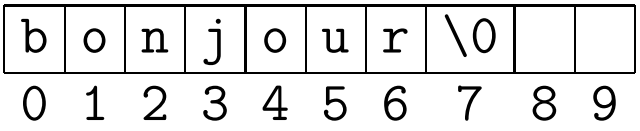
Le résultat est 1. Par contre, l'exécution de

```
int main(void){
  int a[100];
  a[1000]=1;
  return 0;
}
```

provoque une erreur à l'execution ('erreur de segmentation').

Chaînes de caractères

Une chaîne de caractères est un tableau de caractères terminé par le caractère `NUL` ('\0', entier zéro).



L'initialisation peut se faire par

```
char salut[10]="bonjour";
```

ou aussi

```
char *s="bonjour";
```

Attention: une chaîne initialisée comme `s` ci-dessus n'est plus modifiable.

La bibliothèque string.h

La bibliothèque `<string.h>` contient des fonctions sur les chaînes de caractères comme

```
char *strcpy(char *destination,
                  const char *source)

int strcmp(char *s,char *t)

char *strcat(char * destination,
           const char *source);
int strlen(const char *s);
```

Pour les utiliser, on doit ajouter en tête de fichier

```
#include <string.h>
```

Recopie

Voici une fonction qui copie une chaîne de caractères.

```
void copie(char s[], char t[]) {
  int i=0;
  do  {
    s[i] = t[i];
    i++;
  } while (t[i] != '\0')
}
```

Ou encore, de façon plus compacte

```
void copie(char s[], char t[]) {
 int i=0;
 while (s[i] = t[i])
   i++;
}
```

On peut aussi utiliser

```
char *strcpy(char *destination,
                const char *source);
```

Concaténation

Pour concaténer deux chaînes,

```
void concat(char s[], char t[]){
     int i=0, j=0;
     while(s[i]!='\0')
        i++;
     while(t[j] !='\0'){
        s[i+j]=t[j];
        j++;
     }
     s[i+j]='\0';
}
```

on peut aussi utiliser

```
char *strcat(char * destination,
          const char *source);
```

Ainsi, avec

```
char d[50]="Bonjour";
char *s="a tous";
char *p;
```

après l'instruction

```
p=strcat(d,s);
```

le tableau d contient "bonjour a tous" et le pointeur p pointe sur d.

Voici un exemple de tableau à deux dimensions de caractères: les noms des jours de la semaine.

```
char Nom[7][10]={
 "lundi",
 "mardi",
 "mercredi",
 "jeudi",
 "vendredi",
 "samedi",
 "dimanche",
};
```

La fonction ci-dessous donne le nom du n-ième jour:

```
void Lettres(int n) {
  printf("%s\n",Nom[n]);
}
```

Recherche d'un mot dans un texte

L'algorithme suivant recherche le mot x dans le texte y. Il rend le premier indice j de y tel que $y_j \cdots = x_0 \cdots x_n$ et -1 si on ne l'a pas trouvé..

```
int naif(char *x, char *y){
  int i=0,j=0;
  do{
    if(x[i]==y[j]){
      i++;j++;
    }
    else{
      j=j-i+1;
      i=0;
    }
  }while(x[i] && y[j]);
  if(x[i]==0) return j-i else return -1;
}
```

Cours 7
Structures

Une structure est, comme un tableau, un type construit. Avec une structure, on peut grouper dans une seule variable plusieurs variables de types différents. Ceci permet de créer des objets complexes.

Chaque objet possède ainsi des *champs* qui ont eux-même un type. Si p est une structure ayant des champs x,y,z on note p.x,p.y,p.z ces differentes valeurs.

L'affectation d'une structure recopie les champs.

Une structure peut être passée en paramètre d'une fonction et rendue comme valeur. On

pourra par exemple grouper une chaîne de caractères et un entier pour former une fiche concernant une personne (nom et âge).

```
struct fiche{
  char nom[10];
  int age;
};
struct complexe x;
```

Les fonctions

```
void Lire(struct fiche *x){
  scanf(``%s'',x->nom);
  scanf(``%d'',&x->age);
}
void Ecrire(struct fiche x){
  printf(``nom : %s\n
           age : %d\n'',x.nom,x.age);
}
```

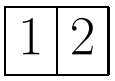
permettent de lire et d'ecrire une fiche.

Nombre complexes

On peut ainsi déclarer un type pour les nombres complexes

```
struct complexe{
  float re;
  float im;
};

struct complexe z;

z.re = 1;
z.im = 2;
```

initialise $z = 1 + 2i$.



On peut ensuite ecrire des fonctions comme

```
struct complexe Somme(struct complexe u,
                      struct complexe v) {
  struct complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}

void afficher(struct complexe z){
  printf(``%f+ %fi\n'',z.re,z.im);
}
```

Exemple

On peut calculer le module d'un nombre complexe par la fonction:

```
float module(struct complexe z){
  float r;
  r= z.re*z.re + z.im*z.im;
  return sqrt(r);
}
```

Puis l'inverse:

```
struct complexe inverse(struct complexe z){
  struct complexe w;
  float c;
  float module = module(z);
  if (module != 0){
    c = module * module;
    w.re= z.re/c;
    w.im= -z.im/c;
    return w;
  }
  else exit(1);
}
```

Usage de typedef

On peut créer un alias pour un nom de type comme:

```
typedef struct {
  float re;
  float im;
} Complexe;
```

On obtient ainsi un alias pour le nom du type.

```
Complexe Somme(Complexe u,Complexe v) {
  Complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}
```

La règle est que le nom de type défini vient à la place du nom de la variable.

On peut ainsi définir

```
typedef int Tableau[N][N];
```

et ensuite:

```
Tableau a;
```

Adresses

Pour représenter une adresse postale:

```
typedef struct {
  int num;
  char  rue[10];
  int code;
  char  ville[10];
} Adresse;
```

```
Adresse Saisir(void) {
  Adresse a;
  printf("numero:"); scanf("%d",&a.num);
  printf("rue:"); scanf("%s",a.rue);
  printf("code:");scanf("%d",&a.code);
  printf("ville:"); scanf("%s",a.ville);
  return a;
}
```

```
void Imprimer(Adresse a) {
  printf("%d rue %s\n%d %s\n",
           a.num,a.rue,a.code,a.ville);
}

int main(void) {
  Adresse a = Saisir();
  Imprimer(a);
  return 0;
}
```

Execution:

```
numero: 5
rue: Monge
code: 75005
ville: Paris

5 rue Monge
75005 Paris
```

Utilisation de plusieurs fichiers

En général, on utilise plusieurs fichiers séparés pour écrire un programme. Cela permet en particulier de créer des bibliothèques de fonctions réutilisables.

Pour l'exemple des nombres complexes, on pourra par exemple avoir un fichier `complexe.c` contenant les fonctions de base et les utiliser dans un autre fichier à condition d'inclure un entête qui déclare les types des fonctions utilisées.

On pourra créer un fichier `entete.h` et l'inclure par `#include "entete.h"`.

Le fichier complexe.c

```
#include "Entete.h"
void Ecrire(Complexe z){
  printf("%f+i%f\n",z.re,z.im);
}
Complexe Somme(Complexe u,Complexe v) {
  Complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}
Complexe Produit(Complexe u, Complexe v){
  Complexe w;
  w.re=u.re*v.re-u.im*v.im;
  w.im=u.re*v.im+u.im*v.re;
  return w;
}
float module(Complexe z){
  float r;
  r= z.re*z.re + z.im*z.im;
  return sqrt(r);
}
```

Le fichier Entete.h

```
#include <stdio.h>
#include <math.h>


typedef struct {
  float re;
  float im;
} Complexe;

void Ecrire(Complexe z);

Complexe Somme(Complexe u, Complexe v);

Complexe Produit(Complexe u, Complexe v);

float module(Complexe z);

Complexe inverse(Complexe z);
```

Le fichier principal

```
#include "Entete.h"


Complexe Exp(float teta){
  //calcule exp(i*teta)
  Complexe w;
  w.re=cos(teta);
  w.im=sin(teta);
  return w;
}
int main(void){
  Ecrire(Exp(3.14159/3));
  return 0;
}
```

On compile par

```
gcc -lm complexe.c exp.c
```

Résultat

```
0.500001+i0.866025
```

Cours 8
Appels récursifs

Une fonction peut en appeler une autre. Pourquoi pas elle-même?

```
int Puissance(int x,int n) {
 int y;

   if (n == 0)
    y = 1;
   else
    y = x * Puissance (x,n-1);
   return y;
}

int main(void) {
  printf("%d\n",Puissance(2,10));
  return 0;
}
```

Résultat : 1024

Récursivité

Un appel récursif est la traduction d'une *définition par récurrence*.

Par exemple, pour la fonction puissance, $y_n = x^n$ est défini par $y_0 = 1$, puis pour $n \geq 1$ par:

$$y_n = \begin{cases} 1 & \text{si } n = 0 \\ x \times y_{n-1} & \text{sinon} \end{cases}$$

Un autre exemple

La fonction factorielle

$$n! = n(n-1)\dots 1$$

est définie par récurrence par $0! = 1$ puis

$$n! = n \times (n-1)!$$

Programmation

```
int Fact(int n) {
  return (n == 0) ?  1 : n * Fact(n-1);
}
```

Comment ça marche?

La mémoire est gérée de façon *dynamique* : une nouvelle zone est utilisée pour chaque appel de la fonction dans la *pile d'exécution*. De cette façon, les anciennes valeurs ne sont pas écrasées.

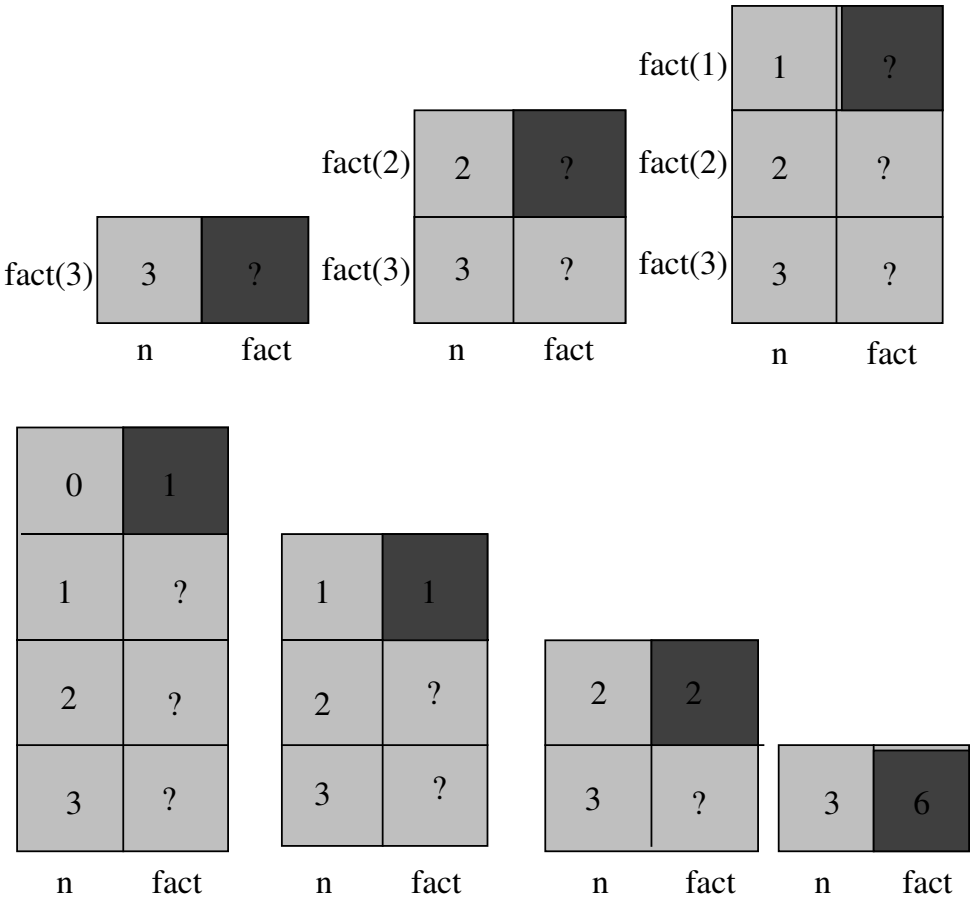


Figure 1: La pile d'execution

Encore un exemple

La suite de Fibonacci est définie par $f_0 = f_1 = 1$ et pour $n \geq 1$

$$f_{n+1} = f_n + f_{n-1}$$

Le programme recursif ci-dessous traduit directement la définition :

```
int Fibo(int n)
{
  return (n==0 || n==1)?
     1 : Fibo(n-1)+Fibo(n-2);
}
```

Mais cette fois le résultat est catastrophique : il faut plus de 10mn sur mon Mac pour calculer (en long) Fibo(30) alors que le calcul est immédiat en itératif.

En fait, le temps de calcul est passé d'une fonction linéaire en n (en itératif) à une exponentielle en n (en récursif).

La représentation binaire

La représentation binaire d'un nombre entier x est la suite $\text{bin}(x) = (b_k, \ldots, b_1, b_0)$ définie par la formule:

$$\begin{aligned} x &= b_k 2^k + \cdots + b_1 2 + b_0 \\ &= 2(b_k 2^{k-1} + \cdots + b_1) + b_0 \end{aligned}$$

ce qui s'écrit aussi

$$\text{bin}(x) = (\text{bin}(q), r)$$

si q, r sont le quotient et le reste de la division entière de x par 2 :

$$x = 2q + r$$

On va voir qu'il est *plus facile* d'écrire le programme sous forme récursive: c'est une simple traduction des relations de récurrence.

Programme récursif

La fonction suivante écrit un nombre entier en binaire.

```
void Binaire (int x) {
   if (x<=1)  printf("%d",x);
   else {
     Binaire(x/2);
     printf("%d",x%2);
   }
}
```

ou encore plus court:

```
void Binaire(int x) {
  if (x >= 2) Binaire(x/2);
  printf("%d", x%2);
}
```

La multiplication du paysan russe

Par exemple, pour calculer 135×26, on écrit

135	26
270	13
540	6
1080	3
2160	1
3510	

A gauche on multiplie par 2 et à droite, on divise par 2. On somme la colonne de gauche sans tenir compte des lignes où le nombre de droite est pair. L'écriture récursive est très simple.

```
int mult(int x, int y){
 if(y==0) return 0;
 return x*(y%2) + mult(x*2,y/2);
}
```

C'est une variante de la décomposition en base 2.

Le jeu des chiffres

On cherche à réaliser une somme s donnée comme somme d'une partie des nombres $w_0, w_1, \ldots, w_{n-1}$.

On remarque qu'on peut se ramener à étudier deux cas:

1. il y une solution utilisant w_0, et donc une solution pour obtenir $s - w_0$ avec $w_1, \ldots, w_{n-1}$.
2. il y a une solution pour obtenir s avec $w_1, \ldots, w_{n-1}$.

De façon générale, le problème $(s, 0)$ se ramène à $(s - w_0, 1)$ ou $(s, 1)$.

On considère donc le problème (s, k) de savoir si on peut atteindre s avec $w_k, \ldots, w_{n-1}$. La fonction suivante rend 1 si c'est possible et 0 sinon.

```
int Chiffres(int s, int k) {
  if (s == 0) return 1;
  if (s < 0 || k>=n) return 0;
  if (Chiffres(s-w[k],k+1) return 1;
  else return Chiffres(s,k+1);
}
```

Moralité

Pour utiliser la récursivité, on doit, en général se poser plusieurs questions:

1. quelles sont les variables du problème?
2. quelle est la relation de récurrence?
3. quels sont les cas de démarrage?

L'utilisation de la récursivité permet d'écrire en général plus facilement les programmes : c'est une traduction directe des définitions par récurrence.

Cependant, les performances (en temps ou en place) des programmes ainsi écrits sont parfois beaucoup moins bonnes.

On peut dans certains cas commencer par écrire en récursif (prototypage) pour optimiser ensuite.

Cours 9
Algorithmes numériques

1. Types numériques en C
2. Arithmétique flottante
3. Calcul de polynômes
4. Le schéma de Horner
5. Calcul de x^n

Représentation des nombres réels

Représentation en *virgule flottante* (on dit aussi notation scientifique):

$$6.02 \times 10^{23}$$

En général la représentation interne d'un nombre réel x se compose de:

1. La *mantisse* m comprise dans un intervalle fixe (comme $[0, 1]$)
2. L'*exposant* e qui définit une puissance de la base b.

qui sont tels que

$$n = m \times b^e$$

Le *nombre de chiffres significatifs* est le nombre de chiffres de la mantisse (prise en base 10). Ainsi, si $b = 2$ et que m a k bits, on a $\lfloor k \log_{10}(2) \rfloor$ chiffres significatifs.

Intuitivement, c'est le nombre de décimales exacts d'un nombre de la forme $0, \cdots$.

Le standard IEEE

1. simple précsion: 32 bits
2. double précision: 64 bits
3. quadruple précision: 128 bits

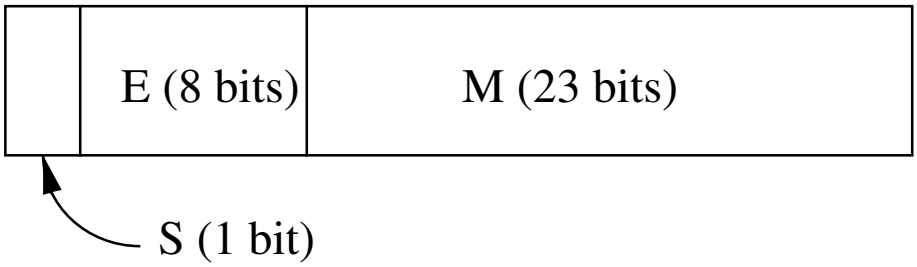


Figure 2: Simple précision

1. Le premier bit est le **bit de signe**
2. Les 8 bits suivants sont l'**exposant** interprété en 127-excès
3. Les 23 bits restants forment la **mantisse** M. Elle est interprétée comme $1.M$

Nombre de chiffres significatifs: 23 bits, soit 6 chiffres significatifs puisque $10^6 < 2^{23} < 10^7$.

Exemple: Le mot de 32 bits

$$1\ 10000111\ 10100000000000000000000$$

représente le nombre

$$\begin{aligned} -2^{135-127} \times 1,625 &= -256 \times 1,625 \\ &= -416 \end{aligned}$$

qui se trouve être entier. Si l'exposant est 01111000 qui représente $(120)_{10} - (127)_{10} = (-7)_{10}$ en code $(127)_{10}$-excès, la valeur est:

$$-2^{-7} \times 1.625 \approx (.0127)_{10}$$

Le drôle de monde de l'arithmétique flottante

Les *erreurs d'arrondi* introduisent des comportements surprenants.

1. $n \oplus 1 = n$ dès que n est assez grand devant 1: si $n = m \times b^e$ avec $0 < m < 1$, on a

   $$n \oplus 1 = b^e(m \oplus b^{-e}) = n$$

   dès que e est plus grand que le nombre de décimales de m. Par exemple, en C,

   ```
   float x=1.0e+8;
   x=x+1;
   printf(``%f\n'',x);
   ```

   affiche 100000000.000000.

2. La série $1 + 1 + 1 + \ldots$ converge.

3. L'addition n'est plus associative. Ainsi

   $$\begin{aligned}(1113. \oplus -1111.) \oplus 7.511 &= 2.000 \oplus 7.511 \\ &= 9.511\end{aligned}$$

alors que

$$\begin{aligned} 1113. \oplus (-1111. \oplus 7.511) &= 1113. \oplus -1103. \\ &= 10.00 \end{aligned}$$

Types numériques en C

On a en C trois types pour représenter des nombres:

1. Le type `int` pour les entiers.
2. Le type `float` pour les réels en simple précision.
3. Le type `double` pour les réels en double précision.

Les constantes de type réel s'écrivent sous la forme

```
3.1415  ou  87.45e+8
```

Le compilateur effectue les *conversions de type* nécessaires. Par exemple, la valeur de l'expression

```
2.0+1
```

est de type réel.

Formats

Pour lire ou écrire un nombre, on a les fonctions usuelles `scanf` et `printf` avec la syntaxe `printf("..",`*exp*) et `scanf("..",`*variable*) où le premier argument est un *format*. Il contient des caractères ordinaires et des spécifications de *conversion*. Chacune est de la forme % suivi d'un caractère.

- %d pour les entiers.
- %f pour les réels.

Des options permettent de préciser le format. Par exemple
`printf("%3d",n)` pour imprimer sur au moins trois caractères.
`printf("%.5f,x)` pour imprimer 5 décimales.

Exemples

Programme:

```
...
x= 10.999;
printf("%f\n",x);
printf("%25f\n",x);
printf("%25.5\n",x);
printf("%25.1f\n",x);
```

Résultat:

```
10.999000
                 10.999000
                  10.99900
                       11.0
```

Calcul de polynômes

On utilise des polynômes

$$p(x) = a_n x^n + \ldots + a_1 x + a_0$$

pour:

1. représenter des entiers:

$$253 = 2 \times 10^2 + 5 \times 10^1 + 3$$

2. approximer des fonctions

$$sin(x) \approx \frac{x^5}{120} - \frac{x^3}{6} + x$$

3. tracer des courbes

4. ...

Programme en C

On suppose que le polynome $p(x)$ est représenté par le tableau

```
float a[N];
```

qui donne les coefficients

$$a[0], a[1], \ldots, a[N-1]$$

On pourra par exemple écrire

```
void EcrirePoly(float p[]){
  int i;
  printf("%.1f",p[0]);
  for(i=1;i<N;i++)
    printf("+%.1fx^%d",p[i],i);
  printf("\n");
}
```

L'exécution de

```
int main(void){
  float a[N]={1,0,1,0};
  EcrirePoly(a);
  return 0;
}
```

produit

```
1.0+0.0x^1+1.0x^2+0.0x^3
```

On peut aussi utiliser une *structure* qui conserve le degré du polynôme comme une information supplémentaire.

```
typedef struct  {
  float coeff[N];
  int degre;
    } Poly;
```

Pour entrer le polynome $x^3 + 2x - 1$, on écrit

```
#define N 20


Poly p;
p.degre = 3;
p.coeff[3] = 1;
p.coeff[2] = 0;
p.coeff[1] = 2;
p.coeff[0] = -1;
```

et cette fois

```
void Ecrire(Poly p){
  int i;
  printf("%.1f",p.coeff[0]);
```

```
  for( i=1 ; i <= p.degre; i++)
    printf("+%.1f x^%d",p.coeff[i],i);
  printf("\n");
}
```

Opérations sur les polynomes

On a sur les polynomes les opérations de somme et de produit. Commençons par la somme. Si

$$\begin{aligned} p(x) &= a_n x^n + \cdots + a_1 x + a_0 \\ q(x) &= b_n x^n + \cdots + b_1 x + b_0 \end{aligned}$$

on a

$$p(x)+q(x) = (a_n+b_n)x^n+\cdots+(a_1+b_1)x+(a_0+b_0)$$

La somme se calcule de la façon suivante :

```
void  Somme(float p[],float q[], float r[]){
 int i;
 for (i= 0; i<N; i++)
   r[i]= p[i]+q[i];
}
```

L'exécution de

```
int main(void){
  float a[N]={1,0,1,0}; float b[N]={0,1,-1,0};
  float c[N];
  Somme(a,b,c);
  Ecrire(c);
```

```
  return 0;
}
```

produit

```
1.0+1.0x^1+0.0x^2+0.0x^3
```

ou encore, avec l'autre structure de données

```
Poly  somme(Poly p, Poly q) {
  int i;
  Poly r;
  r.degre= (p.degre>q.degre)? p.degre:q.degre;
  for (i= 0; i<= r.degre; i++)
    r.coeff[i]= p.coeff[i]+q.coeff[i];
  for (i= r.degre; (i>=0) && (r.coef[i] == 0); 
    r.degree--;
  return r;
}
```

Pour le produit, on a

$$p(x)q(x) = c_{2n}x^{2n} + \cdots + c_1 x + c_0$$

avec

$$c_k = \sum_{i+j=k} a_i b_j$$

Pour le calcul, on suppose que le degré de pq est au plus égal à $N - 1$.

```
void produit(float p[], float q[], float r[]) {
 int i,k,s;
```

```
 for( k= 0; k<N; k++) {
   s =0;
   for (i=0; i<=k ; i++)
     s =s+p[i]*q[k-i];
   r[k]=s;
 }
}
```

Avec l'autre structure de données, on obtient une fonction plus compliquée.

```
Poly produit(Poly p, Poly q) {
  int i,j,k;
  Poly r;
  r.degre= p.degre+q.degre;
  for (k = 0; k<= r.degre; k++) {
    r.coef[k] = 0;
  }
  for (i = 0; i<= p.degre; i++) {
    for (j = 0; j<= q.degre; j++) {
       r.coef[i+j] += p.coef[i] * q.coef[j];
    }
  }
  return r;
```

```
}
```

Evaluation en un point

On veut évaluer le polynôme

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

au point $x = k$.

On veut que le calcul soit:

1. rapide
2. direct: sans consulter une table de valeurs.

Première méthode

On évalue la somme en calculant pour $i = 1, \ldots, n$:

1. La puissance k^i
2. Le produit $a_i k^i$

Pour que le calcul de k^i soit rapide, on aura avantage à calculer **de droite à gauche**:

$$1, k, k^2, \ldots, k^n$$

La méthode naïve

Le calcul de la valeur de $p(x)$ pour $x = k$ se fait par la fonction suivante:

```
float eval(float a[], float k) {
 float val,y ;
 int i;
  val= 0;
  y= 1;
  for (i= 0; i<N; i++)
  {
    val= val + a[i]*y;
    y = y*k;
  }
 return val;
}
```

Deuxième méthode: le schéma de Horner

On fait le calcul en utilisant la factorisation:

$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$$

Le calcul se fait **de gauche à droite**.

Pour $i = n-1, \ldots, 1, 0$ on fait:

1. multiplier par k
2. ajouter a_i

Programme en C

```
float horner(float a[], float k) {
  float val;
  int i;
  val = a[N-1];
  for (i = N-2; i>=0; i--)
     val = val*k+a[i];
 return val;
}
```

Comparaison des méthodes

1. Nombre d'opérations:

	Naïve	Horner
additions	n	n
multiplications	$2n$	n

2. La méthode de Horner ne nécessite pas de connaître les indices i des coefficients a_i: application au calcul de la valeur d'un nombre écrit en base k.

3. La méthode de Horner a une définition *récursive*:

$$p(x) = q(x)x + a$$

Grands nombres

Pour utiliser des nombres de taille arbitraire, il est facile d'écrire des fonctions qui réalisent les opérations de base. On pourra par exemple déclarer

```
typedef struct nombre{
  char chiffres[N];
  int taille;
} Nombre;
```

puis, pour lire caractère par caractère.

```
void Lire(Nombre *x){
  int i=0; char c; char tab[N];
  do{
    scanf("%c",&c);
    tab[i] = c;
    i = i+1;
  } while('0'<= c && c <= '9');
  x->taille = i;
  for(i=0;i<x->taille;i++)
    x->chiffres[i] = tab[x->taille-i-1];
}
```

```
void Ecrire(Nombre x){
  int i;
  for(i=x.taille; i>0;i--)
    printf("%c",x.chiffres[i-1]);
  printf("\n");
}
// chiffre to int
int c2i (char c){
  if ('0'<= c  && c <= '9') return c-'0';
  else return -1;
}

int Somme(Nombre x, Nombre y, Nombre *az){
  int i,t,r=0;

  for(i=0;
      (i<x.taille || i<y.taille || r!=0);
      i++)
  {
   if (i<x.taille && i<y.taille)
      t= c2i(x.chiffres[i])+
         c2i(y.chiffres[i])+ r;
```

```
    else if (i<x.taille)
       t= c2i(x.chiffres[i])+ r;
    else if (i<y.taille)
       t= c2i(y.chiffres[i])+ r;
    else
       t = r;
    r=t/10;
    if (i == N) return 1; // addition impossible
    else az->chiffres[i]=t%10+'0';
  }
  az->taille = i;
  return 0;
}
```

Evaluation de x^n

Pour évaluer des polynômes particuliers, on peut parfois aller plus vite que dans le cas général.

Ainsi le monôme x^n ne nécessite pas n multiplications.

Exemple:

x^{13} peut se calculer en 5 multiplications:

1. on calcule $x^2 = x \times x$
2. on calcule $x^4 = x^2 \times x^2$
3. on calcule $x^8 = x^4 \times x^4$
4. on calcule $x^{12} = x^8 \times x^4$
5. on obtient $x^{13} = x^{12} \times x$

Cette méthode revient à écrire l'exposant en base 2.

Progamme en C

Avec utilisation de la récursivité: on calcule la décomposition en base 2 sous forme récursive (utilisant en fait un schéma de Horner).

```
float puissance (float x, int n) {
  float y;
  if (n== 0) return 1;
  else if (n== 1) return x;
  else if (n%2 == 1) return x*puissance(x,n-1);
  else {
    y = puissance(x,n / 2);
    return y*y;
  }
}
```

Calcul de complexité

Soit $T(n)$ le nombre de multiplications effectués.
On a $T(0) = T(1) = 1$.

On a pour tout entier n pair ou impair

$$T(n) \leq 2 + T(n/2)$$

On va montrer par récurrence que ceci entraine

$$T(n) \leq 2 \log_2 n$$

Ceci est vrai pour $n = 1$. Ensuite, on a

$$\begin{aligned} T(n) &\leq 2 + T(n/2) \\ &\leq 2 + 2 \log_2(n/2) = 2 + 2 \log_2 n - 2 \\ &\leq 2 \log_2 n \end{aligned}$$

Cours 10
Résolution d'équations

1. Généralités
2. Dichotomies
3. La méthode de Newton
4. Equations différentielles

Généralités

On va étudier quelques méthodes de résolution d'équations de la forme:

$$f(x) = 0$$

où f est une fonction (polynôme ou autre) et x une inconnue réelle.

On se place dans le cas où on ne peut pas obtenir de formule explicite pour x. On en cherche une approximation obtenue en appliquant une *itération*:

$$x_{n+1} = g(x_n)$$

ou

$$x_{n+1} = g(x_n, x_{n-1})$$

Pour programmer, on utilise des fonctions de la bibliothèque `math` comme `fabs(),sin(),...` Il faut mettre dans l'en tête `#include math.h` et compiler avec l'option `>gcc -lm fichier.c`.

Dichotomies

C'est la méthode la plus simple. Elle s'applique dès que f est *continue* (i. e. que ses valeurs n'ont pas de saut).

On part de deux valeurs x_1, x_2 telles que

$$f(x_1)f(x_2) < 0$$

et on pose $x_3 = (x_1+x_2)/2$. Si $f(x_1)f(x_3) < 0$, on pose $x_4 = (x_1 + x_3)/2$, sinon on pose $x_4 = (x_3 + x_2)/2$.

L'erreur $e_n = \mid x_n - x_{n-1} \mid$ vérifie:

$$e_n < e_1/2^n$$

On dit que la méthode est *linéaire* : cela signifie en pratique que le nombre de décimales exactes augmente proportionnellement à n.

Programme

On choisit une précision ϵ et on teste la longueur de l'intervalle courant $[a, b]$.

```
float dicho(float a, float b, float eps){
 float x,fa,fx;

  fa = f(a);
  do {
   x = (a+b)/2;
   fx = f(x);
   if (fa*fx < 0)  b = x;
   else {
      a = x; fa = fx;
    }
  } while ((b-a) > eps);
 return x;
}
```

Variante

Si la fonction f est croissante, on peut écrire de façon plus simple:

```
float dicho(float a, float b, float eps) {
 float x,fa,fx;

  do {
    x = (a+b)/2;
    fx = f(x);
   if (fx > 0)
     b = x;
    else
     a = x;
  } while ((b-a) > eps);
  return x;
}
```

Fonctions en paramètres

Les fonctions en C ne sont pas des variables mais on peut utiliser des pointeurs sur des fonctions comme paramètres de procédure.

Par exemple :

```
float dicho(float a, float b,
            float eps, float (*f) (float)){
  float x;

  do {
    x = (a+b)/2;
   if ((*pf)(x) > 0)  b = x; else a = x;
  }
  while ((b-a) > eps);
  return x;
}
```

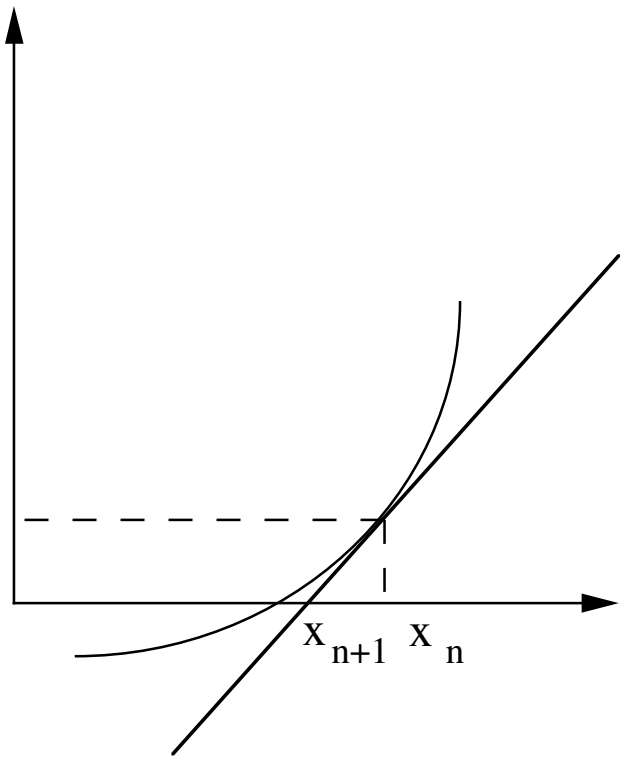
On peut ensuite passer à la fonction une fonction en paramètre:

```
#include <math.h>

float f(float x) {
  return sin(x)-0.5;
}



int main(void) {
  printf("%f",dicho(0,1.5,1e-6, &f));
  return 0;
}
```

Suite des valeurs de x
0.750000 0.375000 0.562500 0.468750 0.515625
0.539062 0.527344 0.521484 0.524414 0.522949
0.523682 0.523315 0.523499 0.523590 0.523636
0.523613 0.523602 0.523596 0.523599 0.523600
0.523599 0.523599

La méthode de Newton

Cette méthode (aussi appelée méthode de la tangente) s'applique quand la fonction f est dérivable.



Formule:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Calcul de l'erreur

On suppose f suffisamment dérivable. Posons

$$F(x) = x - f(x)/f'(x)$$

On calcule en dérivant:

$$F'(x) = 1 - \frac{f'^2(x) - f(x)f''(x)}{f'^2(x)} = \frac{f(x)f''(x)}{f'^2(x)}$$

Si la racine r est simple, on a $F'(r) = 0$ et $F"(r) = -f''(r)/f'(r)$ d'où l'erreur $e_n = x_n - r$:

$$e_{n+1} = \frac{F''(\xi_n)}{2} {e_n}^2$$

On dit que c'est une méthode dont la convergence est quadratique : cela signifie en pratique que le nombre de décimales exactes double en passant de n à $n + 1$.

Programmation

Il suffit de garder la dernière valeur x_n dans une variable x. On suppose que les valeurs de f et f' sont données par des fonctions C `f` et `g`.

```
float newton(float a, float eps,
             float (*f) (float), float (*g) (fl
 float x,y;

  x = a;
  do {
    y = x;
    x = x-f(x)/g(x);
  } while (fabs(x-y) > eps);
 return x;
}
```

Exemple

Si on applique la méthode de Newton à la fonction $f(x) = x^2 - a$, on obtient une méthode de calcul de $\sqrt{a}$.

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

```
float newton(float a, float eps) {
  float x,y;

  x =a;
  do
  {
   y = x;
   x = (x+(a/x))/2;
  } while (fabs(x-y) > eps);
  return x;
}
```

Programme complet

```
#include <math.h>

float  u, e;
int  n;

float f(float x) {
 return x*x-2;
}

float g(float x) {
 return 2*x;
}
```

```
float dicho(float a, float b,
            float eps, int * pi)
{
  float x,fa,fx;

  fa = f(a); *pi=0;
  do {
    x = (a+b)/2;
    fx = f(x); (*pi)++;
    if (fa*fx < 0)
      b = x;
    else {
      a = x; fa = fx;
    }
  }
  while ((b-a) > eps);
 return x;
}
```

```
float newton(float a, float eps, int * pi) {
  float x,y;

  x = a; *pi=0;
  do {
    y = x; (*pi)++;
    x = x-f(x)/g(x);
  }
  while (fabs(x-y) > eps);
  return x;
}
```

```
int main(void)
{
 printf("u=");
 scanf("%f",&u);
 e=1e-6;
 printf("methode des dichotomies\n");
 printf("racine de %2f =%9.8f obtenue
     en %2d etapes\n",u,dicho(0,10,e,&n),n);
 printf("racine de %2f =%9.8f obtenue
     en %2d etapes\n",u,newton(u,e,&n),n);
 return 0;
}
```

Execution

```
u = 2

methode des dichotomies :
 racine de 2.0e+0 = 1.41421378
 obtenue en 24 etapes

methode de Newton :
 racine de 2.0e+0 = 1.41421354
 obtenue en 5 etapes
```

La valeur de $\sqrt{2}$ avec 9 décimales est

$$1.414213562$$

Table

n	Dichotomies	Newton
1	5.00000000	**1.**50000000
2	2.50000000	**1.41**666663
3	**1.**25000000	**1.41421**568
4	**1.**87500000	**1.41421354**
5	**1.**56250000	**1.41421354**
6	**1.4**0625000	
7	**1.4**8437500	
8	**1.4**4531250	
9	**1.4**2578125	
10	**1.41**601562	
11	**1.41**113281	
12	**1.41**357422	
13	**1.414**79492	
14	**1.414**18457	
15	**1.414**48975	
16	**1.414**33716	
17	**1.4142**6086	
18	**1.4142**2272	
19	**1.4142**0364	
20	**1.41421**318	
21	**1.41421**795	
22	**1.41421**556	
23	**1.41421**437	
24	**1.41421**378	

Equations différentielles

Une équation différentielle est une équation de la forme

$$y' = f(x, y)$$

où f est une fonction continue de x et y. Une solution de l'équation dans un intervalle I de R est une fonction

$$y : x \mapsto y(x)$$

telle que pour tout $x \in I$ on ait

$$y'(x) = f(x, y(x)).$$

On démontre, sous des hypothèses sur la fonction f, qu'il existe une unique solution satisfaisant la *condition initiale*

$$y(x_0) = y_0$$

Exemples

L'équation

$$y' = ay$$

a pour solution $y = e^{ax}$. L'équation

$$y' = a$$

a pour solutions

$$y = ax + b$$

pour une constante b à choisir.

La méthode d'Euler

On choisit un *pas* h et on pose

$$x_n = x_0 + hn, \quad y_{n+1} = y_n + hf(x_n, y_n)$$

pour une condition initiale (x_0, y_0), de telle sorte que y_n est une approximation de $y(x_n)$. On démontre que l'erreur

$$e_n = |y_n - y(x_n)|$$

tend vers 0 avec h sous des hypothèses appropriées sur f.

Programmation

```
float Euler (float x0,float y0, float h,
                float (* f) (float,float)){
  int i;
  float x, y;

  x = x0;
  y = y0;
  while (x < 1) {
    y = y + h * (*f)(x, y);
    x = x + h;
   }
  return y;
}
```

Programme complet

Si on choisit $f(x, y) = y$, on écrit:

```
 float Euler (float x0,float y0, float h,
        float (*f)(float,float));
 float fy (float x, float y){
  return y;
}

int main(void){
 int i;
  float h;

  h = 0.1;
  for (i = 1; i<=5; i++) {
    printf("%6.6f",Euler( 0, 1, h, fy));
    h = h / 10;
  }
  return 0;
}
```

Résultats

L'erreur est de l'ordre de $h = 10^{-i}$. Mais à partir de $h = 10^{-5}$, les erreurs d'arrondi s'accumulent et le calcul diverge.

i	Euler
1	**2**.593743
2	**2.7**31862
3	**2.71**9637
4	**2.718**143
5	**2.71**5568

Si on passe en double précision (il suffit de déclarer `double x,y;` dans Euler), la dernière ligne devient

5	**2.71828**95

qui est bien exacte à 10^{-5} près.

Cours 11
Systèmes d'équations linéaires

1. La méthode d'élimination
2. L'algorithme de Gauss
3. Programmation
4. Choix du pivot

Systèmes d'équations linéaires

Les systèmes *linéaires* comme:

$$\begin{aligned} x + 3y - 4z &= 8 \\ x + y - 2z &= 2 \\ -x - 2y + 5z &= -1 \end{aligned}$$

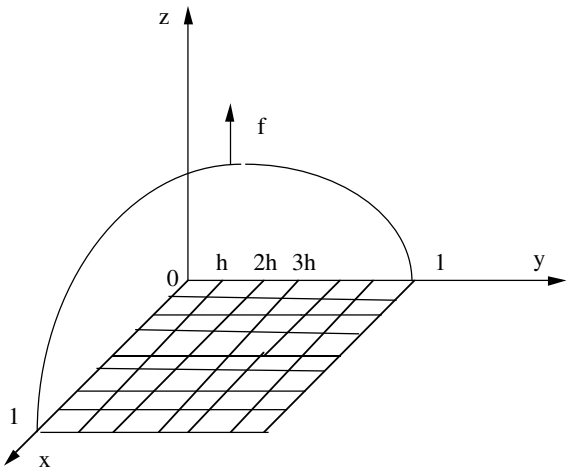
sont très fréquents dans les applications en:

1. physique:
   (a) électricité
   (b) élasticité
   (c) accoustique
2. chimie
3. économie

En informatique, on veut résoudre des *grands* systèmes linéaires insolubles à la main: le nombre de variables va jusqu'au million ou plus.

Exemple en calcul de structures

On représente une plaque élastique par une fonction $p(x, y)$ donnant la hauteur au point (x, y).



Si la plaque est soumise à une force $f(x, y)$ alors p est solution de l'équation

$$\Delta p + \lambda f = 0$$

où $\Delta p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$ et λ est le coefficient d'élasticité.

On *discrétise* le domaine en choisissant $h = 1/N$ et en calculant

$$p_{ij} = p(ih, jh)$$

pour $0 \leq i, j \leq N$.

Les $(N+1)^2$ inconnues p_{ij} sont solutions du système d'équations linéaire donné par

$$\Delta p_{ij} = f_{ij}$$

où Δ est l'opérateur associant à p la fonction $q = \Delta p$ dont la valeur au point M est

$$q(M) = 4p(M) - \sum_{M' \text{ voisin de } M} p(M')$$

Pour $h = 10^{-3}$ on aura 10^6 inconnues.

Exemple

Considérons le système de trois équations à trois inconnues:

$$\begin{aligned} x + 3y - 4z &= 8 \\ x + y - 2z &= 2 \\ -x - 2y + 5z &= -1 \end{aligned}$$

ou encore:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ x_1 + x_2 - 2x_3 &= 2 \\ -x_1 - 2x_2 + 5x_3 &= -1 \end{aligned}$$

ou sous forme matricielle:

$$\begin{bmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \\ -1 \end{bmatrix}$$

On élimine x_1 de la deuxième équation en la remplacant par la première moins la deuxième:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ 2x_2 - 2x_3 &= 6 \\ -x_1 - 2x_2 + 5x_3 &= -1 \end{aligned}$$

De même, en remplaçant la troisième par la somme de la première et de la troisième, on élimine x_1 de la troisième:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ 2x_2 - 2x_3 &= 6 \\ x_2 + x_3 &= 7 \end{aligned}$$

Si on remplace enfin la troisième par la difference entre la deuxième et deux fois la troisième, on obtient le système:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8 \\ 2x_2 - 2x_3 &= 6 \\ -4x_3 &= -8 \end{aligned}$$

qui est sous forme *triangulaire*. On obtient la solution facilement: on tire de la dernière la valeur de x_3:

$$x_3 = 2$$

En reportant dans la deuxième, on obtient:

$$2x_2 - 4 = 6$$

d'où $x_2 = 5$. Enfin en reportant dans la première, on obtient

$$x_1 + 15 - 8 = 8$$

d'où $x_1 = 1$.

Le cas général

On part d'un système de n équations à n inconnues:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + \ldots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{10}x_0 + a_{11}x_1 + \ldots + a_{1,n-1}x_{n-1} &= b_1 \\ &\vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned}$$

qu'on écrit aussi:

$$\begin{bmatrix} a_{00} & a_{01} & \ldots & a_{0,n-1} \\ a_{10} & a_{11} & \ldots & a_{1,n-1} \\ \vdots & & & \\ a_{n-1,0} & a_{n-1,1} & \ldots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

L'algorithme d'élimination

1. *démarrage*: On part avec $k = 0$ de la première inconnue x_0.

2. *itération* A la k-ième étape, on élimine x_k des $n - k - 1$ dernières équations.

   Pour cela, on change pour chaque indice i $(k < i \leq n - 1)$, la ligne i en

   $$l_i - (a_{ik}/a_{kk})l_k$$

   Ainsi a_{ik} est changé en:

   $$a_{ik} - (a_{ik}/a_{kk})a_{kk} = 0$$

3. *résultat*: Le système est sous forme triangulaire.

Figure

La formule magique est :

$$a_{ij} \longleftarrow a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$$

Programmation

On simplifie la programmation en considérant le second membre comme une $(n+1)$-ième colonne. On déclare le système comme:

```
float a[N][N+1];
```

Puis on a le coeur de l'algorithme:

```
void trianguler(void){
 int k,i,j;
 float r;

 for (k = 0; k<N-1;k++)
  for (i = k+1; i<N; i++){
    r= a[i][k]/a[k][k];
    for (j = k+1; j<=N; j++)
     a[i][j] = a[i][j] - r*a[k][j];
   }
}
```

La substitution

On calcule la solution du système triangulaire par *retour arrière* :

```
void substituer(void){
  int j, k;
  float t;

  for (j = N-1; j>=0; j--) {
    t = 0.0;
    for (k = j + 1; k<N; k++)
        t = t + a[j][ k] * x[k];
    x[j] = (a[j][ N] - t) / a[j][j]
  }
}
```

Fichier principal

```
#define N 3

float a[N][N+1];
float x[N];

void trianguler(void);
void substituer(void);
void lire(void);
void ecrire(void);

int main(void){
 lire();
 trianguler();
 substituer();
 ecrire();
 return 0;
}
```

Temps de calcul

On va compter le nombre $T(n)$ d'opérations arithmétiques $(+, \times, /)$ utilisées pour résoudre un système de n équations par cette méthode. C'est une mesure du temps de calcul.

Pour la triangulation, il faut, pour chaque valeur de k, effectuer $n-k$ divisions, $(n-k)(n-k+1)$ additions et autant de multiplications. On a donc:

$$\begin{aligned} T(n) &= \sum_{k=1}^{n} (2(n-k+1)(n-k) + (n-k)) \\ &= 2\sum_{i=1}^{n} i(i-1) + \sum_{i=1}^{n} i = \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i \\ &= \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \\ &= \frac{2}{3}n^3 + \ldots \end{aligned}$$

La substitution n'ajoute qu'un polynôme de degré 2 en n.

*Cls *.c onclusion*: le temps de calcul est un polynôme en n^3. Pour $n = 10^3$ une machine à 10Mflops prendra $0.6 \times 10^2 s \approx$ 1mn.

Problème de choix du pivot

On ne peut pas appliquer la formule magique

$$a_{ij} \leftarrow a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$$

si le pivot a_{kk} est nul.

Même s'il n'est pas nul mais tres petit, le calcul donne un mauvais résultat. Ainsi, si on part du système:

$$\begin{aligned} 10^{-4}x + y &= 1 \\ x + y &= 2 \end{aligned}$$

et que l'on calcule (en flottant) avec 3 décimales, on obtient par triangulation le système:

$$\begin{aligned} 10^{-4}x + y &= 1 \\ (-0.999 \times 10^4)y &= (-0.999 \times 10^4) \end{aligned}$$

de solution $y = 1, x = 0$ alors que la solution du système de départ est

$$x = 1.00010\ldots, y = 0.99990\ldots$$

Stratégie du pivot maximal

Si le système d'équations a une solution unique, on ne peut pas trouver au cours de l'algorithme de colonne nulle: la valeur de la variable correspondante serait indéterminée.

On peut donc toujours *échanger* les équations k et i (avec $i > k$) de façon à avoir un pivot $a_{kk} \neq 0$.

En fait on a avantage à rechercher systématiquement le pivot maximal pour éviter les petits pivots (autant que possible).

```
void trianguler(void){
  int i, j, k, max;
  float   t;
  for (k = 0; k<N-1; k++)
  {
    max = k;
    for (i= k+1; i<N; i++)
     if (fabs(a[i][k]) > fabs(a[max][k]))
         max = i;
    for (j = k; j<=N; j++){
      t = a[k][j]; a[k][j] = a[max][j];
      a[max][j] = t;
    }
        /*comme avant*/
  }
}
```

Cours 12
Programmation linéaire

1. Introduction
2. L'algorithme du simplexe
3. Programmation
4. Ennuis
5. Conclusion

Un exemple économique

On utilise trois matières premières I, II et III pour fabriquer deux produits A et B.

	Produits A	Produits B	Quant. disponible
I	2	1	8
II	1	2	7
III	0	1	3
Quant. produite	x_1	x_2	

On cherche de plus à rendre maximum le profit

$$4x_1 + 5x_2$$

Le problème à résoudre est donc:

Maximiser $4x_1 + 5x_2$
sous les contraintes :

$$x_1 \geq 0, x_2 \geq 0$$
$$2x_1 + x_2 \leq 8$$
$$x_1 + 2x_2 \leq 7$$
$$x_2 \leq 3$$

Représentation graphique

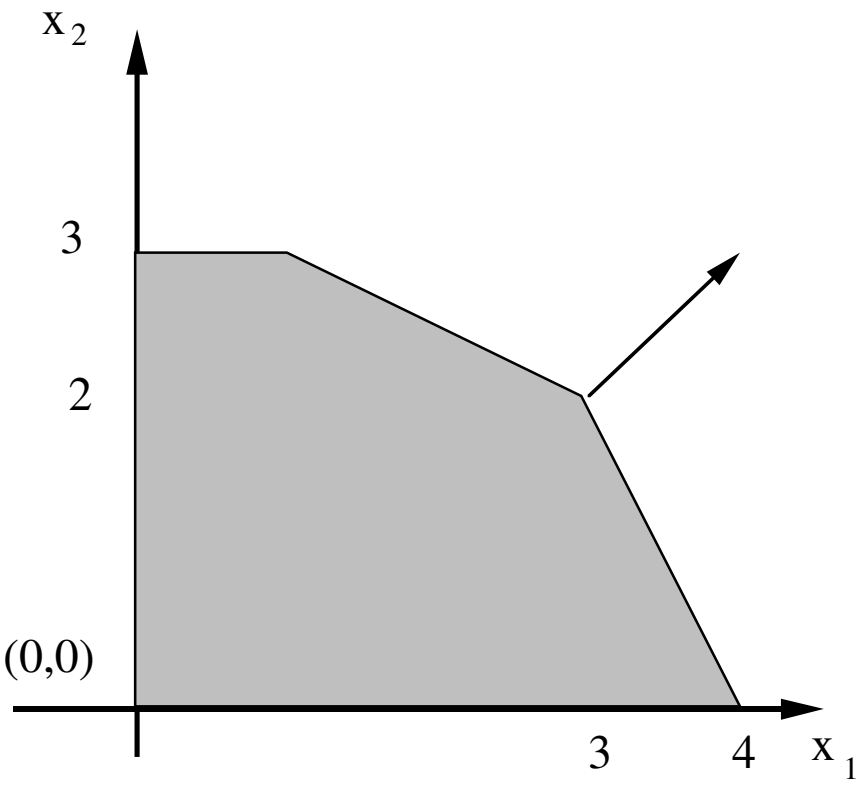


Figure 3: Le simplexe

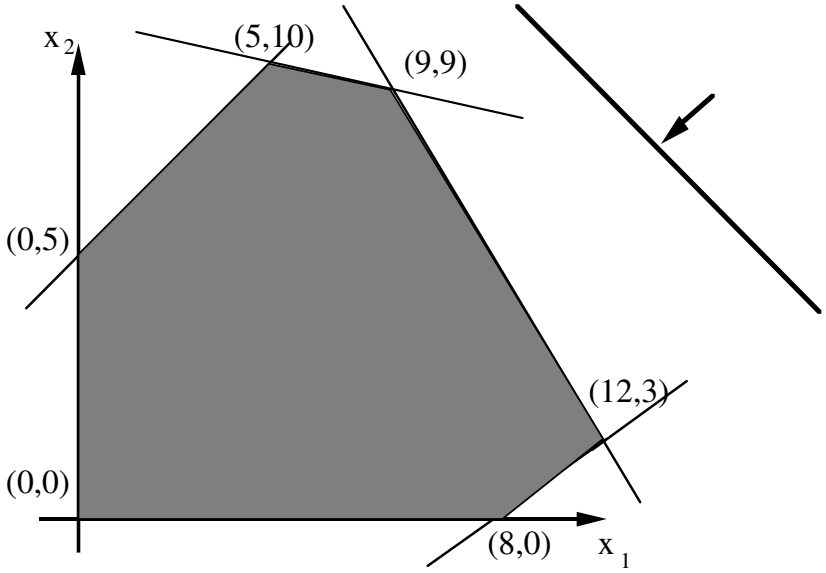
Un exemple simple

On considère le problème suivant.

Maximiser $x_1 + x_2$
sous les contraintes:

$$\begin{aligned} -x_1 + x_2 &\leq 5 \\ x_1 + 4x_2 &\leq 45 \\ 2x_1 + x_2 &\leq 27 \\ 3x_1 - 4x_2 &\leq 24 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Interprétation graphique



la méthode du simplexe

On met le système de départ sous la forme:

$$\begin{aligned} -x_1 + x_2 + y_1 &= 5 \\ x_1 + 4x_2 + y_2 &= 45 \\ 2x_1 + x_2 + y_3 &= 27 \\ 3x_1 - 4x_2 + y_4 &= 24 \end{aligned}$$

ou encore:

$$\begin{aligned} y_1 &= 5 + x_1 - x_2 \\ y_2 &= 45 - x_1 - 4x_2 \\ y_3 &= 27 - 2x_1 - x_2 \\ y_4 &= 24 - 3x_1 + 4x_2 \end{aligned}$$

On augmente autant qu'on peut x_1: sa valeur est donnée par la quatrième équation:

$$x_1 = 8$$

On élimine ensuite x_1 en tirant sa valeur de la

quatrième équation:

$$\begin{aligned} y_1 &= 13 + \frac{1}{3}x_2 - \frac{1}{3}y_4 \\ y_2 &= 37 - \frac{16}{3}x_2 + \frac{1}{3}y_4 \\ y_3 &= 11 - \frac{11}{3}x_2 + \frac{2}{3}y_4 \\ x_1 &= 8 + \frac{4}{3}x_2 - \frac{1}{3}y_4 \end{aligned}$$

avec la fonction d'objectif:

$$8 + \frac{7}{3}x_2 - \frac{1}{3}y_4$$

On élimine cette fois x_2 en le tirant de la troisième équation.

$$\begin{aligned} y_1 &= 14 - \frac{1}{11}y_3 - \frac{3}{11}y_4 \\ y_2 &= 21 + \frac{16}{11}y_3 - \frac{7}{11}y_4 \\ x_2 &= 3 - \frac{3}{11}y_3 + \frac{2}{11}y_4 \\ x_1 &= 12 - \frac{4}{11}y_3 - \frac{1}{11}y_4 \end{aligned}$$

avec

$$z = 15 - \frac{7}{11}y_3 + \frac{1}{11}y_4$$

Comme y_4 a un coefficient positif dans z, on cherche à l'augmenter: sa valeur maximum est 33 donnée par la deuxième équation.

En substituant on obtient:

$$\begin{aligned} y_1 &= 5 - \frac{5}{7}y_3 + \frac{3}{7}y_2 \\ y_4 &= 33 - \frac{11}{7}y_2 + \frac{3}{7}y_3 \\ x_2 &= 9 - \frac{2}{7}y_2 + \frac{1}{7}y_3 \\ x_1 &= 9 \end{aligned}$$

En substituant dans z, on obtient finalement:

$$z = 18 - \frac{1}{7}y_2 - \frac{3}{7}y_3$$

d'où la solution:

$$x_1 = 9, x_2 = 9$$

Formulation générale

On part d'un sytème sous la forme d'un *dictionnaire*:

$$\text{Maximiser} \sum_{j=1}^{n} c_j x_j$$

sous les contraintes

$$\begin{aligned} \sum_{j=1} a_{ij} x_j &\leq b_j \quad (1 \leq i \leq m) \\ x_j &\geq 0 \quad (1 \leq j \leq n) \end{aligned}$$

que l'on met sous la forme:

$$\begin{aligned} x_{n+i} &= b_i - \sum_{j=1} a_{ij} x_j \quad (1 \leq i \leq m) \\ z &= \sum_{j=1}^{n} c_j x_j \end{aligned}$$

en introduisant les nouvelles variables

$$x_{n+1}, \ldots, x_{n+m}$$

Principe de l'algorithme

- *Initialisation*: on part d'un dictionnaire.
- *Itération*: tant qu'il existe un variable x_j à droite avec un coefficient positif dans z
  1. on la prend comme variable entrante
  2. on cherche la variable sortante x_k: c'est celle qui est à gauche dans l'équation qui bloque la première l'augmentation de x_j.
  3. on exprime x_j dans l'équation définissant x_k et on reporte dans les autres.

Programmation Maple

Maple est un système de calcul formel (il y en a d'autres).

On peut programmer la méthode du simplexe par des commandes de très haut niveau:

```
>with(simplex):
cnsts:={-x+y<=5, x+4*y<=45,
2*x+y<=27,3*x-4*y<=24}:
obj:=x+y:
maximize(obj,cnsts union {x>=0,y>=0});
```

$$\{x = 9, y = 9\}$$

Programmation en C

On représente le dictionnaire (N équations et M variables) par un tableau

`a[N][M]`

avec deux conventions:

1. La ligne 0 est la fonction z (avec des signes inversés).
2. La colonne M est le second membre.

Ainsi le système:

$$\begin{aligned} -x_1 + x_2 &\leq 1 \\ 2x_1 - x_2 &\leq 2 \end{aligned}$$

avec la fonction d'objectif:

$$z = x_1 + x_2$$

est représenté par la matrice:

$$\begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & 1 \\ 2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

La fonction pivot

Si q est la variable entrante et p la variable sortante, la substitution est effectuée par la fonction:

```
void pivot(int p, int q)
{
 int j,k;

 for (j = 0; j<=N; j++)
  for (k = M; k>=0; k--)
   if (j != p && k != q)
    a[j][k] = a[j][k]-a[p][k]*a[j][q]/a[p][q];
 for (j= 0; j<=N; j++)
        if (j != p)  a[j][q] = 0;
 for (k = 0; k<=M; k++)
     if (k != q)  a[p][k] = a[p][k]/a[p][q];
 a[p][q] =1;
}
```

C'est l'algorithme de Gauss!

L'algorithme du simplexe lui-même s'écrit alors:

```
int main void)
{
 do
  for(q = 0; q < M && a[0][q]>= 0; q++);
  for(p = 0; p < N && a[p][q] <= 0; p++);
   for (i = p+1; i<N; i++)
   if a[i,q]>0 then
    if (a[i][M]/a[i,q] < a[p][M]/a[p,q])
                 p= i;
  if (q<M && p < N)  pivot(p,q);
 while (q< M && p< N);
 return 0;
}
```

Ennuis

On peut avoir trois sortes de problèmes avec l'algorithme du simplexe.

1. Initialisation: comment choisir le point de départ?
2. Itération: comment choisir les variables entrante et sortante?
3. Terminaison: comment être sûr de s'arrêter?

Initialisation

Pour trouver un point de départ, on considère le *problème auxiliaire*:

Minimiser x_0

sous les contraintes:

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i \quad (1 \leq i \leq m)$$

$$x_j \geq 0 \quad (0 \leq j \leq n)$$

Ce problème a une solution de départ: il suffit de prendre x_0 assez grand.

Le problème de départ a une solution ssi le problème auxiliaire a une solution avec $x_0 = 0$.

Choix du pivot

Il peut y avoir le choix de

1. Plusieurs variables entrantes. On peut choisir celle qui a le coefficient maximum dans z
2. Plusieurs variables sortantes Ceci peut conduire à un cycle.
3. Aucune variable sortante. On dit alors que le problème est *non-borné*: il n'y a pas de valeur maximale de z.

Terminaison

La méthode du simplexe peut boucler indéfiniment. On peut l'éviter en utilisant une *stratégie* de choix des variables sortantes.

Même si elle ne boucle pas, la méthode peut, dans des cas rares, prendre un temps *exponentiel* en fonction du nombre de variables (penser au nombre de sommets possibles d'un polyèdre dans l'espace à n dimensions).

Conclusion

La méthode du simplexe résoud un problème très fréquent dans les applications: celui de l'optimisation linéaire.

Il existe beaucoup d'autres méthodes:

1. En optimisation non-linéaire
2. Pour des cas particuliers de l'optimisation linéaire: réseaux de transport
3. Pour remplacer le simplexe: algorithmes de Khachian, Karmarkar, etc.