

# INFORMATIQUE SCIENTIFIQUE

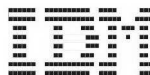
IUT ORSAY

Mesures Physiques - 1<sup>re</sup> année  
Tome 2 - TP 5 à 8

version 3.0

<http://www.iut-orsay.fr/dptmphy/Pedagogie/Welcome.html>

Bob CORDEAU



## **Informatique :**

Rencontre de la logique formelle et du fer à souder.

Maurice NIVAT

### **SOMMAIRE**

- TP 5 : APPLICATIONS MATHÉMATIQUES DES FONCTIONS
- TP 6 : NUAGE DE POINTS ET DROITES DE RÉGRESSION
- TP 7 : LES POINTEURS
- TP 8 : MANIPULATIONS D'OCTETS

## APPLICATIONS MATHÉMATIQUES DES FONCTIONS

Tout passe.  
L'art robuste seul a l'éternité.  
Le buste survit à la cité.  
Théophile GAUTIER

*Objectifs :*

- introduction d'analyse numérique ;
- où l'on constate l'importance des algorithmes !

Fichiers à produire : TP5e1.cpp TP5e2.cpp TP5e3.cpp TP5e4.cpp

## 1 La dichotomie

La *dichotomie* (mot qui signifie de l'on partage en deux parties) est un procédé assez simple et efficace pour déterminer, avec une marge d'erreur prévue à l'avance, la solution d'une équation à l'intérieur d'un intervalle donné dans lequel on sait pertinemment qu'il en existe exactement une. S'il existait plusieurs solutions dans cet intervalle, la dichotomie permettrait d'approcher l'une d'entre elles sans que l'on puisse facilement prévoir laquelle.

Cette méthode n'a pas la réputation d'être rapide. Toutefois, contrairement à d'autres plus performantes en temps, elle permet à coup sur d'atteindre une solution approchée. En outre, la borne supérieure de l'erreur globale tient compte ici des *erreurs machines*.

### 1.1 Principe : *diviser pour trouver !*

On se place dans la situation suivante :

$f$  est une fonction définie, continue et monotone dans l'intervalle  $[V_{inf}; V_{sup}]$ , avec la condition :

$$f(V_{inf}) \times f(V_{sup}) < 0$$

Cette condition est facile à comprendre, elle signifie que  $f(V_{inf})$  et  $f(V_{sup})$  n'ont pas le même signe : la courbe doit passer d'un côté à l'autre de l'axe des abscisses. Comme cette courbe est continue, elle doit couper l'axe des abscisses... donc l'équation  $f(x) = 0$  possède une solution et la monotonie assure que cette solution est unique.

Pour trouver un encadrement de largeur  $p$  ( $p$  comme précision) de la solution, on pratique comme suit :

- Tant que l'écart entre  $V_{inf}$  et  $V_{sup}$  est supérieur à  $p$ , on calcule le nombre

$$essai = \frac{V_{inf} + V_{sup}}{2}$$

et on teste le signe de  $f(V_{inf}) \times f(V_{sup})$  :

- Si  $f(V_{inf}) \times f(essai) > 0$ , c'est qu'ils ont le même signe, donc la solution cherchée est entre  $essai$  et  $V_{sup}$  : on remplace  $V_{inf}$  par  $essai$  et on recommence...
- Si  $f(V_{inf}) \times f(essai) < 0$ , c'est qu'ils n'ont pas le même signe, donc la solution cherchée est entre  $V_{inf}$  et  $essai$  : on remplace  $V_{sup}$  par  $essai$  et on recommence...

- Si  $f(V_{inf}) \times f(essai) = 0$ , c'est que  $f(essai) = 0$  : on remplace  $V_{inf}$  par  $essai - p/2$  et  $V_{sup}$  par  $essai + p/2$ .<sup>1</sup>
- et pour terminer, on rend les valeurs modifiées de  $V_{inf}$  et  $V_{sup}$ .

**En résumé :**

Comme l'intervalle de recherche est partagé en deux, si la solution cherchée n'est pas dans une « moitié », c'est qu'elle est dans l'autre !

Informatiquement, il s'agit de transcrire ce principe en un algorithme d'une procédure `Dicho()` qui reçoit trois arguments : les deux bornes de l'intervalle initial de recherche (qui seront modifiées par la procédure) et l'erreur tolérée (qui, elle, reste inchangée).

**Attention**

☞ Rédiger cette procédure en langage algorithmique et écrire son prototype C.

**1.2 Essai de la procédure `Dicho()` (fichier TP5e1.cpp)**

Écrire un programme principal qui saisie les deux limites de l'intervalle initial, la borne supérieure de l'erreur, puis qui affiche la racine de la fonction  $f(x) = \cos(x) - \exp(x)/2$  (cf. figure 1).

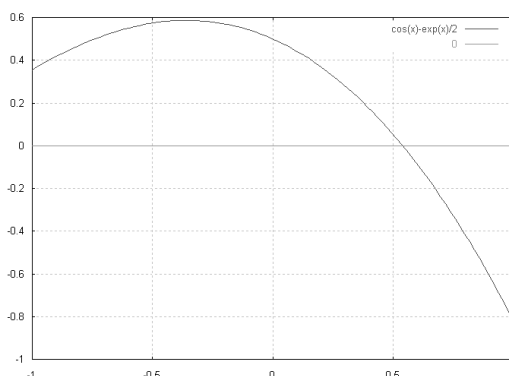


FIG. 1 – Graphe de la fonction  $f(x) = \cos(x) - \exp(x)/2$

Pour se donner une idée de l'intervalle judicieux de recherche, il est utile de tracer la fonction  $f(x)$ . GnuPlot est un *grapheur* qui permet de tracer rapidement des courbes mathématiques. Ouvrez GnuPlot et affichez la courbe en tapant simplement :

```
plot [-1:1] cos(x)-exp(x)/2
```

où `[-1:1]` représente l'intervalle de tracé.

Il reste à écrire deux éléments :

- définir la procédure `Dicho()` en traduisant votre algorithme ;
- définir la fonction `F()` qui retourne  $\cos(x) - \exp(x)/2$ .

**2 Algorithme de l'intégrale simple**

**2.1 Ce que c'est...**

Une solution possible de calcul d'une intégrale simple sur l'intervalle  $[a, b]$  est d'utiliser la méthode dite *des rectangles*<sup>2</sup>. Elle consiste à calculer l'aire  $S(a, b)$  délimitée par l'axe  $Ox$ , les droites  $x = a$  et  $x = b$ , et la courbe

<sup>1</sup>Parfois on préfère remplacer  $V_{inf}$  par  $essai - p/3$  et  $V_{sup}$  par  $essai + p/3$ .

<sup>2</sup>ou plus exactement « algorithme de Riemann ».

$y = f(x)$ .

Quand on divise  $[a, b]$  en  $n$  intervalles égaux, et que l'on pose  $\Delta x = \frac{b-a}{n}$ , on voit que le produit  $\Delta x \cdot f(a + i\Delta x)$  représente l'aire d'un petit rectangle, c'est-à-dire une valeur approchée de l'aire comprise entre la courbe, l'axe des abscisses et les droites  $x = x_i, x = x_{i+1}$ . La somme algébrique de ces produits constitue une approximation de  $S(a, b)$  d'autant meilleure que  $\Delta x$  est plus petit.

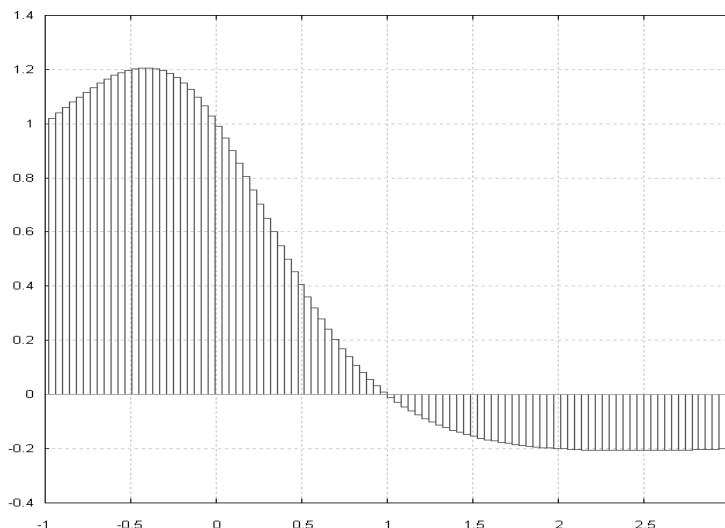


FIG. 2 – Méthode des rectangles

Le passage à la limite de cette somme constitue l'algorithme de Riemann :

$$\int_a^b f(x) dx = \lim_{n \rightarrow \infty} \Delta x \sum_{i=1}^n f(a + i \Delta x)$$

**Attention**

✎ Écrire l'algorithme de la fonction `Integrale()` qui reçoit trois arguments en entrée : les deux bornes de l'intervalle d'intégration et le nombre de pas de la somme<sup>1</sup> et qui retourne la somme algébrique des aires des petits rectangles.

Écrire le prototype de la fonction `Integrale()`.

**2.2 ... application au calcul de  $\pi$  (fichier TP5e2.cpp)...**

Transcrire votre algorithme en C pour créer la fonction `Integrale()`.

Créer aussi une fonction à intégrer qui retourne  $\frac{1}{1+x^2}$  et qui correspond au prototype suivant : `double F(double x)` ; Entre quelles bornes  $a$  et  $b$  doit-on intégrer `F()` pour obtenir un résultat dont il soit facile de déduire la valeur de  $\pi$  ?

Écrire enfin le programme principal qui saisit le nombre de pas d'intégration, appelle la fonction `Integrale()` avec les arguments voulus et affiche une approximation de  $\pi$ .

Remplissez le tableau suivant :

Nombre de pas d'intégration	10	1000	100000
Approximation de $\pi$			
Erreur relative à M.PI			

<sup>1</sup>Une optimisation simple et efficace de la formule de Riemann consiste à remarquer que la fonction à intégrer est *linéaire*, et qu'il est donc inutile de faire  $n$  multiplications...

### 2.3 ... et au calcul de $e$ (fichier TP5e3.cpp)

On souhaite calculer une valeur approchée de  $e$  (base du logarithme népérien) en utilisant l'algorithme de Riemann. On cherche la borne  $b$  telle que :

$$I = \int_1^b \frac{dx}{x} = 1$$

En effet, si  $b = e$ , alors  $I = 1$ . Comme de plus la fonction logarithme est continue et monotone, on est fortement tenté de faire varier  $b$  en utilisant la dichotomie pour approcher  $e$  avec la précision souhaitée.

**À faire :**

- définir la fonction  $F(x)$  à intégrer qui retourne  $\frac{1}{x}$  ;
- copier la fonction `Integrale()` précédente ;
- écrire une fonction `Dicho2()`, proche de la précédente<sup>1</sup>, mais avec les modifications suivantes dans la boucle `do .. while` :
  - calculez la borne courante :  $b = b_1 + \frac{b_2 - b_1}{2}$  où  $b_1$  et  $b_2$  représentent un encadrement de la borne supérieure d'intégration et affichez sa valeur ;
  - calculez  $I = \int_1^b F(x)$  en  $10^5$  pas ;
  - si  $I < 1$ , la borne inférieure est trop petite et faites  $b_1 = b$  ;
  - sinon la borne supérieure est trop grande et faites  $b_2 = b$  ;
  - bouclez tant que l'écart entre les bornes est supérieur à  $10^{-4}$ .
- enfin écrire le programme principal qui se contente de saisir un encadrement de la borne supérieure de l'intégrale (c'est-à-dire un encadrement de  $e$ ). Imposez  $b_1 < 2$  et  $b_2 > 3$ . Appelez la fonction `Dicho2()` dont on fournit le prototype<sup>2</sup> et affichez l'approximation trouvée sous la forme (cf. figure 3) :

$$x \times 10^{-3} < e < (x + 1) \times 10^{-3}$$

```

Entrez un encadrement de 'e' :  bMin = 1
                               bMax = 5

b = 3
b = 2
b = 2.5
b = 2.75
b = 2.625
b = 2.6875
b = 2.71875
b = 2.70313
b = 2.71094
b = 2.71484
b = 2.7168
b = 2.71777
b = 2.71826
b = 2.71851
b = 2.71838
b = 2.71832

==> 2718.10-3 < e < 2719.10-3
    
```

FIG. 3 – Approximation de  $e$  par dichotomie

### 3 Un utilitaire (fichier TP5e4.cpp)

Pour terminer ce TP, nous allons développer un petit programme qui nous sera utile pour le TP n° 7.

<sup>1</sup>il aurait été possible de réutiliser la fonction précédente, mais ici le problème nécessite une adaptation tout en gardant complètement l'esprit de cette méthode.

<sup>2</sup>`double Dicho2(double, double); //(entrée, entrée)`

Il s'agit d'écrire deux fonctions. La première se nomme `RandInt()` et fournit un entier aléatoire entre deux bornes entières passées en argument.

La seconde, un peu plus complexe, se nomme `RandDouble()` et fournit un double aléatoire entre deux bornes de type `double` et pour une précision (de type `double`) passée en troisième argument.

Dans le `main()` du fichier `TP5e4.cpp`, déclarez (entre autres) :

- une constante entière  $N$  valant 15;
- un tableau d'entiers, `tInt[]`, de taille  $N$ ;
- un tableau de doubles, `tDouble[]`, de taille  $N$ .

Saisissez les bornes inférieures et supérieures pour `RandInt()`, et dans une boucle, affectez et affichez le tableau `tInt[]` (cf. figure 4).

```

N nombres aleatoires entiers dans [a .. b] :
      a = 1
      b = 5

tInt[0] = 3
tInt[1] = 5
tInt[2] = 5
tInt[3] = 2
tInt[4] = 5
tInt[5] = 5
tInt[6] = 1
tInt[7] = 3
tInt[8] = 1
tInt[9] = 5
tInt[10] = 1
tInt[11] = 2
tInt[12] = 3
tInt[13] = 4
tInt[14] = 3

```

FIG. 4 – Fonction `RandInt()`

De même, saisissez les bornes inférieures et supérieures et la précision pour `RandDouble()`, et dans une boucle, affectez et affichez le tableau `tDouble[]` (cf. figure 5).

```

N nombres aleatoires flottants dans [x .. y] :
      x = 1.1
      y = 3.2
      precision : 1e-3

tDouble[0] = 2.2
tDouble[1] = 2.256
tDouble[2] = 1.64
tDouble[3] = 2.648
tDouble[4] = 1.934
tDouble[5] = 3.165
tDouble[6] = 1.712
tDouble[7] = 2.361
tDouble[8] = 1.637
tDouble[9] = 1.391
tDouble[10] = 1.576
tDouble[11] = 3.012
tDouble[12] = 2.604
tDouble[13] = 2.879
tDouble[14] = 2.416

```

FIG. 5 – Fonction `RandDouble()`

Enfin écrire les définitions des deux fonctions.

## CE QU'IL FAUT RETENIR

- ☞ En analyse numérique l'algorithme est *primordial* ;
- ☞ Dans ce domaine on n'utilise généralement que des `double` (seuls flottants connus par la bibliothèque mathématique) et des `long` ;
- ☞ Dans les conditions des boucles, *ne pas* faire de test d'égalité entre flottants (problème de précision machine), mais des tests d'inégalité.

## 4 À préparer pour le prochain TP

### Attention

☞ Bien connaître votre cours sur les pointeurs, c'est un *point dur* en C, et préparez les exercices ci-dessous.

### 4.1 Définition des pointeurs

Pour le langage C, une variable possède trois caractéristiques :

- un nom (par exemple *i*).
- une valeur (par exemple 5).
- une adresse dans la mémoire de l'ordinateur, qu'on appellera pour simplifier, pointeur sur cette variable.

Pour déclarer un pointeur *p* sur un entier, il suffit d'écrire :

```
int * p;
```

### 4.2 Les opérateurs à propos des pointeurs

Récupérer la valeur d'une variable à partir de son nom est très simple. Ainsi pour afficher la valeur d'une variable, il suffit d'écrire :

```
// Declaration et initialisation de la variable i
int i=5;
// Affichage de la valeur de variable i
cout<<i;
```

#### 4.2.1 L'opérateur &

Cependant, le programmeur a besoin dans certains cas de récupérer le pointeur (c'est-à-dire l'adresse) sur une variable. Pour récupérer le pointeur sur la variable *i*, on procède comme suit en utilisant l'opérateur & :

```
// Declaration et initialisation de la variable i
int i;
i=5;

// Declaration d'un pointeur sur un entier
int * p;

// On met dans p le pointeur sur la variable i
p=&i;
```

Dans ce cas, *p* ne vaut pas 5, mais vous indique à quel endroit dans la mémoire la valeur de la variable *i* est stockée.

#### 4.2.2 L'opérateur \*

Dans d'autres cas, le programmeur n'a accès qu'à un pointeur, et n'a donc pas de nom de variable. Pour afficher la valeur sur laquelle pointe *p*, il suffit d'utiliser l'opérateur \* de la façon suivante :

```
// Declaration et initialisation de la variable i
int i;
i=5;

// Declaration d'un pointeur sur un entier
int * p;

// p pointe sur la variable i
p=&i;
```

```
// affichage de la valeur sur laquelle pointe le pointeur p, c'est-a-dire 5
cout << *p;
```

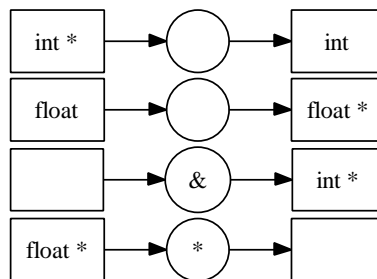
### 4.3 Conclusion

L'opérateur & nous donne, à partir d'une variable (par exemple de type *int*), son pointeur (de type *int \**). L'opérateur \* fait l'inverse et nous donne, à partir d'un pointeur *p* (de type *int \**) la valeur (de type *int*) pointée par *p*. En d'autre terme, l'opérateur & transforme une variable de type entier (*int*) en un type pointeur sur entier (*int \**) et l'opérateur \* transforme un type pointeur sur entier (*int \**) en un type entier.

### 4.4 Exercices

#### 4.4.1 Pointeurs

Remplissez les carrés par les bons types et les cercles par les bons opérateurs.



#### 4.4.2 Structures

Réviser vos connaissances sur les structures et les entrées/sorties sur fichiers textes et préparer par écrit les programmes TP6e5.cpp et TP6e6.cpp.

**LES POINTEURS**

Tu tires ou tu pointes ?  
Marcel, bouliste émérite

---

*Objectifs :*

- pointeurs typés ;
- utilisation des pointeurs dans les procédures ;
- les structures.

Fichiers à produire : TP6e1.cpp TP6e2.cpp TP6e3.cpp TP6e4.cpp TP6e5.cpp TP6e6.cpp

---

## 1 Différences entre *fonctions* et *procédures* (fichier TP6e1.cpp)

Écrire le programme TP6e1.cpp en respectant les points suivants :

- écriture des prototypes :
  1. une procédure `Doubler()` qui reçoit un pointeur sur un entier,
  2. une procédure `Incrementer()` qui reçoit un pointeur sur un entier,
  3. une fonction `Tripler()` qui reçoit un entier et retourne un entier.
- écriture du programme principal :
  1. déclaration d'un entier `a` de type `int`,
  2. saisie de `a` ;
  3. appel de `Doubler()`, `Incrementer()` puis `Tripler()` sur `a`,
  4. affichage de `a` ainsi modifié.
- écriture des définitions de `Doubler()`, `Incrementer()` et `Tripler()`.

Testez votre programme.

```
Entrez une valeur entiere : 5
      Etat initial : a = b = 5
Après avoir double, incremente puis triple a, on obtient : a = 33
Après avoir double, incremente puis triple b, on obtient : b = 33
```

FIG. 6 – Procédures élémentaires sur les pointeurs

Modifiez-le alors de la façon suivante :

- ajoutez le prototype d'une procédure `Idem()` qui reçoit un pointeur sur un entier ;
- déclarez un second entier `b` qui recevra la même valeur initiale que `a` ;
- ajoutez un appel à `Idem()` sur `b` puis affichez `b` ainsi modifié ;

- enfin écrivez la définition de `Idem()` qui doit faire les appels à `Doubler()`, `Incrementer()` puis `Tripler()` sur son argument de façon à produire le même résultat (cf. figure 6).

Tester votre programme.

## 2 Calcul sur les nombres complexes (fichier TP6e2.cpp)

Le programme principal effectue la saisie de deux `double a` et `b`, respectivement partie réelle et partie imaginaire d'un nombre complexe cartésien.

On demande d'écrire une procédure `ModArg()` ayant quatre arguments : `a`, `b`, `mod` et `arg`. À partir des deux premiers (c'est-à-dire que `a` et `b` sont des *entrées*), `ModArg()` calcule le module et l'argument en degrés (c'est-à-dire que `mod` et `arg` sont des *sorties*).

Rappels :

- La racine carrée s'écrit `sqrt(x)` et l'arctangente `atan2(y, x)`. Ces fonctions sont déclarées dans `<math.h>`, donc vous n'avez pas à les déclarer, il suffit d'inclure l'en-tête `<math.h>` ;
- la valeur de  $\pi$  est aussi définie dans le même en-tête et s'écrit : `M_PI`.

Le programme principal doit faire un *appel* à cette procédure et afficher les valeurs du module et de l'argument du complexe saisi.

Faites un test sur des valeurs simples, par exemple :

$$a = b = 1$$

## 3 Procédure d'échange (fichier TP6e3.cpp)

On veut écrire une procédure nommée `Swap()` qui reçoit deux entiers et qui les échange. Par exemple, si `a` vaut 10 et `b` vaut 15, après avoir utilisé `Swap()`, `a` vaut 15 et `b` vaut 10.

Quel prototype proposez-vous pour cette procédure ? Réalisez son codage en C.

Écrivez une fonction principale qui demande à l'utilisateur deux valeurs entières `v1` et `v2`, puis qui affiche dans le même ordre (`v1` puis `v2`) ces deux valeurs, d'une part avant l'appel de `Swap()` et d'autre part après. À l'aide de `Swap()`, écrivez une procédure `Range()` qui reçoit deux entiers et qui les range du plus petit au plus grand. Faites un appel de `Range()` puis affichez une dernière fois `v1` et `v2` dans le même ordre.

Faites au moins deux tests :

1. `v1 = 2, v2 = 3`
2. `v1 = 3, v2 = 2`

## 4 Placement aléatoire (fichier TP6e4.cpp)

Le problème consiste à écrire une procédure de mélange d'un tableau trié, qui a l'avantage de réutiliser la procédure `Swap()` précédente. Cette procédure peut être utilisée dans un jeu informatique, par exemple pour effectuer un mélange de cartes...

Le *vrai* problème est que l'on ignore *a priori* la taille du jeu de cartes et qu'on est donc incapable de déclarer le tableau qui l'implémente...

La solution réside dans l'*allocation programmée*. Ce mécanisme permet de fournir un emplacement mémoire à un pointeur. On peut alors déclarer un pointeur d'entiers (au lieu de déclarer un tableau), saisir dans le programme la taille du jeu de cartes et allouer la mémoire nécessaire grâce à la fonction `malloc()` :

```
tab = (int *)malloc(taille*sizeof(int));
```

On voit que `malloc()` s'utilise de la façon suivante :

```
pointeur_sur_zone_mémoire = (type_pointeur)malloc(taille_objet)
```

En fin de programme principal, n'oubliez pas de rendre à l'ordinateur la mémoire que vous lui avez prise, en utilisant la fonction `free()`.

Écrivez un programme principal qui saisie la taille  $T$  du jeu de cartes, alloue le tableau, l'affecte par des entiers de 1 à  $T$  et l'affiche.

Dans une boucle que l'on peut interrompre en appuyant sur la touche `Echap` (code 27), on appelle la procédure `Melange()` et on affiche le résultat (cf. figure 7).

La procédure `Melange()` n'a qu'un argument, le tableau d'entiers. Son rôle, comme vous pouvez le voir sur la figure 7, est de permuter les éléments du tableau, sans en oublier ni créer de doublons.

Proposez un algorithme et codez-le.

```

Avant melange on a le tableau :
~~~~~
 1  2  3  4  5  6  7  8  9 10
~~~~~

Après 1 melange(s) on a le tableau :
 7  4  8 10  6  3  2  9  1  5
      On recommence <"Echap" pour terminer> ?

Après 2 melange(s) on a le tableau :
 1  2  9  6  7  4  5  3 10  8
      On recommence <"Echap" pour terminer> ?

Après 3 melange(s) on a le tableau :
 3  7  5  6  4 10  9  8  2  1
      On recommence <"Echap" pour terminer> ? _

```

FIG. 7 – Mélange d'un tableau d'entiers

## 5 Rappels sur les structures en C

### 5.1 Introduction et notation

Les structures font parties plus généralement des types étiquetés qui comprennent les types énumérés (`enum`), les types structures (`struct`) et les types unions (`union`). Ce sont les seuls types que l'on puisse réellement définir en tant que *nouveaux* types à part entière, possédant un nom propre valable en particulier pour définir des variables.

Nous allons nous restreindre aux types structures. Par rapport aux types tableaux, les structures ont l'avantage de pouvoir regrouper des données *hétérogènes* (de natures différentes). Ainsi, au lieu de gérer une adresse à l'aide de variables séparées :

```

char nom[20];
prenom[20];
unsigned short numeroRue;
char rue[20];
unsigned int codePostal;
char ville[20];

```

doit-on considérer que ces informations sont liées *sémantiquement*<sup>1</sup> et qu'il vaut mieux les traiter comme un tout :

```

struct Adresse
{
  char nom[20];
  prenom[20];
  unsigned short numeroRue;
  char rue[20];
  unsigned int codePostal;
}

```

<sup>1</sup>c'est-à-dire par leur sens.

```
char ville[20];
};
```

## 5.2 Déclaration et accès aux champs

Nous venons de définir un nouveau type structuré et nous pouvons maintenant déclarer des variables de ce type.

Pour affecter les différentes composantes (appelées les *champs*) de la variable `monAdresse`, on utilise la notation pointée, par exemple :

```
// déclaration
struct Adresse monAdresse;
// affectation
strcpy(monAdresse.nom, "Perinet");
strcpy(monAdresse.prenom, "Francois");
monAdresse.numeroRue = 99;
strcpy(monAdresse.rue, "avenue des Champs-Elysees");
monAdresse.codePostal = 75008;
strcpy(monAdresse.ville, "Paris");
```

## 6 Gestion de fichiers ASCII

Lorsqu'un programme doit lire ou écrire des données dans un fichier, il utilise un *tampon d'entrée-sortie* (en anglais *buffer*). Au lieu de gérer directement un fichier, nous allons utiliser ce tampon qui est une zone de mémoire RAM dans laquelle les données sont temporairement stockées. L'emplacement mémoire (donc l'adresse) du tampon d'un certain fichier est fourni par une variable structurée du type `FILE*`, appelée un *flux*. Le type `FILE` est défini en tant que structure dans le fichier d'en-tête `<stdio.h>`. Pour accéder concrètement à un fichier dans votre programme, vous devez déclarer un pointeur vers une variable *fichier* de type `FILE` :

```
// declare un flux
FILE *fichier;
```

Avant qu'un programme puisse manipuler un fichier, il doit commencer par l'ouvrir grâce à la fonction système `fopen()`, dont le prototype est :

```
// ouvre un flux
FILE *fopen(char nom_fichier[], char mode_accès[]);
```

Le premier paramètre est le nom du fichier à ouvrir, il sera contenu dans une variable du type chaîne de caractères (par exemple : `char ficN[40]`), et le second indique la nature des opérations à effectuer sur le fichier ; on a principalement le choix entre les options suivantes :

- r : ouverture en lecture seule ;
- w : création pour écriture. Si le fichier préexiste son contenu est perdu.

On notera que si, pour une raison quelconque, l'ouverture échoue, la fonction `fopen()` retourne la valeur prédéfinie `NULL`.

Une fois le fichier correctement ouvert (et dans le bon mode) on peut effectuer des lectures ou des écritures formatées à l'aide des fonctions : `fprintf()` et `fscanf()`. On les utilise exactement de la même façon que `printf()` et `scanf()`, mais en ajoutant comme premier argument le descripteur de fichier donné par `fopen()`.

Enfin, après avoir lu ou écrit dans un fichier, il faut fermer le flux ouvert par `fopen()` en utilisant la fonction `fclose` avec la syntaxe :

```
// ferme un flux
fclose(fichier);
```

### 6.1 Écriture d'une structure (fichier TP6e5.cpp)

Dans le programme principal :

- déclarez une structure `Figure` comprenant trois champs : une variable `nom` de type `char [40]` et deux variables `a` et `b` de type `double`, représentant les axes d'une ellipse. Déclarez une variable de ce type : `fig_Ecrit`. Déclarez enfin un pointeur de `FILE` : `fichier_Ecrit`.
- saisir le nom et les longueurs des axes de l'ellipse ;
- calculer l'aire de l'ellipse ( $aire = \pi * a * b$ );

- écrivez ces trois informations dans le fichier `figure.txt` (par défaut ce fichier sera stocké dans votre répertoire courant `InfoScient`);
- vérifiez avec le Bloc-notes que votre fichier `figure.txt` est correct.

## 6.2 Lecture d'une structure (fichier `TP6e6.cpp`)

Nous allons maintenant relire ces informations.

Dans le programme principal :

- déclarez la même structure `Figure`. Déclarez une variable de ce type : `fig_Lu`. Déclarez aussi un pointeur de `FILE` : `fichier_Lu`.
- lisez les information dans le fichier `figure.txt`;
- vérifiez l'aire.

## CE QU'IL FAUT RETENIR

- ☞ rappels sur la déclaration et l'utilisation d'un pointeur ;
- ☞ différences entre fonctions et procédures ;
- ☞ passage des pointeurs en argument d'une procédure ;
- ☞ utilisation des structures en C ;
- ☞ gestion simple des fichiers textuels.

## 7 À préparer pour le prochain TP

Écrire en C le programme principal du fichier `TP7e1.cpp`.



## NUAGE DE POINTS ET DROITES DE RÉGRESSION

L'astre des nuits s'avance en chassant les orages :  
 Clarisse, sois pour moi l'astre calme et vainqueur  
 Qui de mon front troublé dissipe les nuages  
 Et fait rêver mon cœur.

François-René de CHATEAUBRIAND (dernière strophe du poème *Clarisse*)

*Objectifs :*

– les entrées/sorties sur fichiers

Fichiers à produire : TP7e1 . cpp

## 1 Droites de régression

### 1.1 Introduction

On rappelle que si un nuage de points donné a une forme *allongée* il existe différentes méthodes pour créer l'équation d'une droite représentant *au mieux* ce nuage. En fait, suivant la méthode, on obtient des droites différentes, à moins que les points ne soient parfaitement alignés, auquel cas toutes les méthodes se valent et donnent toutes l'équation de la droite sur laquelle sont les points du nuage.

On va envisager ici la méthode des *moindres carrés*.

Cette méthode consiste à calculer la somme des carrés des écarts entre les points du nuage et les points d'une droite *estimée* et à choisir comme meilleure droite celle pour laquelle la somme des carrés des écarts est minimum.

Il reste alors à choisir la façon de calculer les *écarts* et il y a deux choix possibles classiques :

- soit pour chaque point du nuage on utilise la différence entre l'ordonnée de ce point et celle du point de la droite qui aurait la même abscisse ;
- soit pour chaque point du nuage on utilise la différence entre l'abscisse de ce point et celle du point de la droite qui aurait la même ordonnée.

En utilisant la première méthode, on obtient la droite de régression de  $y$  en  $x$ , et en utilisant la deuxième la droite de régression de  $x$  en  $y$ .

Le programme à créer doit permettre à l'utilisateur de trouver par approximations successives quelle est la *meilleure* droite de régression de  $y$  en  $x$  pour un nuage choisi de façon aléatoire. <<

### 1.2 Droite de régression de $y$ en $x$ (fichier TP7e1 . cpp)

Nous allons tout d'abord *structurer* les données manipulées. Pour cela, créez une structure DROITE contenant deux `double` :  $a$ , coefficient directeur de la droite, et  $b$ , son terme constant. Créez aussi une structure POINT contenant deux `double` :  $x$  et  $y$ , coordonnées d'un point du plan. Pour représenter le nuage, on définit un type NUAGE comme un tableau de POINT de la façon suivante :

```
typedef POINT NUAGE[NBPTS] ;
```

où NBPTS, nombre de points du nuage, est une constante entière que l'on prendra égale à 20.

Il faut maintenant *concevoir* les différentes fonctions utiles au programme :

- `RempliTNuage()` est une procédure qui a un NUAGE en entrée/sortie et qui ne renvoie rien. Son rôle est de générer NBPTS points aléatoires (dans certaines limites...) et de les stocker dans le fichier textuel

nuage.dat dans votre répertoire de travail. Elle vous est fournie ci-dessous (elle utilise la fonction `RandDouble()`, développée lors du TP n° 5);

- `EcartOrdo()` est une fonction qui reçoit un NUAGE en entrée/sortie, une DROITE en entrée et qui renvoie un double. Elle calcule la somme des carrés des écarts entre les ordonnées des points du nuage et des points de la droite estimée;
- `PointMoyen()` est une fonction qui reçoit le NUAGE en entrée/sortie et qui renvoie un POINT. Elle calcule l'abscisse moyenne et l'ordonnée moyenne du nuage;
- `Variance()` est une procédure qui reçoit un NUAGE en entrée/sortie, deux pointeurs de double en sortie (la variance en  $x$  et la variance en  $y$ ), un POINT en entrée (le point moyen de la droite) et qui ne renvoie rien;
- `CoVariance()` est une fonction qui reçoit un NUAGE en entrée/sortie, un POINT en entrée (le point moyen de la droite) et qui renvoie un double<sup>1</sup>.
- `EcritDroites()` est une procédure qui reçoit un NUAGE en entrée/sortie, un POINT en entrée (le point moyen de la droite) deux double en entrée (la covariance et la variance en  $x$ ), une DROITE (l'estimée) et qui ne renvoie rien. Elle écrit dans le fichier textuel `d_X.dat` les NBPTS points de la droite calculée et dans `d_U.dat` les NBPTS points de la droite estimée.

```
void RemplitNuage(NUAGE n)
{
    // prototype local
    double RandDouble(double, // réel inférieur compris (entrée)
                      double, // réel supérieur compris (entrée)
                      double); // précision (entrée)
    // variables locales
    FILE *f = fopen("nuage.dat", "w");
    double a = RandDouble(-2.0, 2.0, 0.1),
           b = RandDouble(-5.0, 5.0, 0.1),
           aleaX, aleaY;

    for(int i = 0; i < NBPTS; i = i + 1)
    {
        aleaX = RandDouble(-0.1, 0.1, 0.1);
        n[i].x = (double)i + aleaX;
        aleaY = RandDouble(-2.0, 2.0, 0.1);
        n[i].y = a*n[i].x + b + aleaY;
        fprintf(f, "%3.31f\t%3.31f\n", n[i].x, n[i].y);
    }
    fclose(f);
}
```

<sup>1</sup>Voir le rappel de statistique en fin de TP, page 20.

Il reste à écrire le programme principal dont voici l'algorithme :

**DébutProgramme**

**DébutDéclaration**

*estim* : DROITE  
*pMoy* : POINT  
*nimbus* : NUAGE  
*varX*, *varY*, *cov* : flottant  
*rep* : entier

**FinDéclaration**

**Afficher**("Le nuage est choisi par l'ordinateur et est enregistré dans le fichier 'nuage.dat'")

RemplitNuage(*nimbus*)

**Faire**

**Afficher**("Coefficient directeur de la droite d'estimation : ")

**Saisir**(*estim.a*)

**Afficher**("Terme constant pour cette droite : ")

**Saisir**(*estim.b*)

**Afficher**("EcartOrdo() = ", *EcartOrdo(nimbus, estim)*)

**Afficher**("On change de droite (o/n) ? ")

**Saisir**(*rep*)

**TantQue**(*rep* ≠ 'n')

*pMoy* ← *PointMoyen(nimbus)*

*Variance(nimbus, varX, varY, pMoy)*

*cov* ← *coVariance(nimbus, pMoy)*

*EcritDroite(nimbus, pMoy, cov, varX, estim)*

**Afficher**("Pour EcartOrdo() la bonne droite était : ")

**Afficher**("Y = ", *cov/varX*, "\*X + ", *pMoy.y - pMoy.x \* cov/varX*)

**Afficher**("Corr = ", (*cov \* cov*)/(*varX \* varX*))

**FinProgramme**

**Remarque**

Utiliser le grapheur GnuPlot pour visualiser le résultat obtenu. Pour afficher sur un même graphe les deux courbes et le nuage de points, tapez :

```
plot 'd.X.dat' w l, 'd.U.dat' w l, 'nuage.dat' w p
```

**Rappels de statistique :**

- Pour un nuage de points  $M_i(x_i, y_i)$  où  $i$  varie de 0 à  $N - 1$ , on appelle *point moyen* le point de coordonnées :

$$\bar{x} = \sum_{i=0}^{N-1} \frac{x_i}{N} \quad \text{et} \quad \bar{y} = \sum_{i=0}^{N-1} \frac{y_i}{N}$$

- On appelle *variance* d'une suite de réels  $x_i$  (respectivement  $y_i$ ), où  $i$  varie de 0 à  $N - 1$ , les réels :

$$var_x = \sum_{i=0}^{N-1} \frac{(x_i - \bar{x})^2}{N} \quad \text{et} \quad var_y = \sum_{i=0}^{N-1} \frac{(y_i - \bar{y})^2}{N}$$

- On appelle *covariance* de la suite  $(x_i, y_i)$ , où  $i$  varie de 0 à  $N - 1$ , le réel :

$$cov_{xy} = \sum_{i=0}^{N-1} \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}$$

- Équation de la droite estimant  $y$  en fonction de  $x$  :

$$\hat{y} = \frac{cov_{xy}}{var_x} x + (\bar{y} - \bar{x} \frac{cov_{xy}}{var_x})$$

- Le *coefficient de corrélation* est enfin le réel :

$$cor_{xy} = \frac{cov_{xy}^2}{var_x var_y}$$

## CE QU'IL FAUT RETENIR

- ☞ Face à un problème réel (physique ou mathématique), apprendre à *structurer* les données ;
- ☞ maîtriser la notation pointée pour les structures (ou fléchée pour les pointeurs de structure) ;
- ☞ apprendre à *concevoir* les différentes fonctions et procédures pour résoudre le problème posé (analyse *descendante*) ;
- ☞ enfin synthétiser ces éléments dans le programme principal (analyse *ascendante*). Si une donnée est mal structurée ou une fonction mal conçue, reprendre l'analyse au début.

## 2 À préparer pour le prochain TP

- Écrire en C le programme TP8e1 .cpp ;
- écrire en C la procédure AffOct ( ) du TP n° 8 après avoir assimilé la notion de masque et les opérateurs de décalage.



## MANIPULATION D'OCTETS

Être ou ne pas être : voilà la question.

SHAKESPEARE

*Objectifs :*

- utilisation des pointeurs ;
- utilisation des opérateurs de bits.

Fichiers à produire : TP8e1 . cpp TP8e2 . cpp TP8e3 . cpp

### 1 Prérequis (fichier TP8e1 . cpp)

En utilisant votre cours, écrivez un programme qui déclare deux booléens  $a$  et  $b$ . Écrivez une boucle de test des opérateurs binaires sur le modèle suivant :

**DébutProgramme**

**DébutDéclaration**

$a, b$  : booléen

$rep$  : entier

**FinDéclaration**

**Faire**

**Afficher**("Valeur de a : ")

**Saisir**( $a$ )

**Afficher**("Valeur de b : ")

**Saisir**( $b$ )

**Afficher**("a  $\otimes$  b = ",  $a \otimes b$ )

**Afficher**("On recommence (o/n) ? ")

**Saisir**( $rep$ )

**TantQue**( $rep \neq n'$ )

**FinProgramme**

Dans cet algorithme,  $\otimes$  désigne un opérateur binaire parmi :

- le **ET bit à bit** ;
- le **OU bit à bit** ;
- le **OU exclusif bit à bit, noté XOR**.

Vous devrez également tester l'opérateur unaire **NON bit à bit**.

Remarque importante : Pour le langage C, 0 est considéré comme FAUX et tout autre entier est considéré comme VRAI. Par exemple l'octet 0000 0000 = (0)<sub>10</sub> vaut FAUX et l'octet 0100 0010 = (66)<sub>10</sub> vaut VRAI.

Utilisez le programme précédent pour remplir le tableau suivant :

octet1	0	0	1	0	1	1	0	1
octet2	1	1	1	0	0	1	0	1
octet1 <b>ET</b> octet2								
octet1 <b>OU</b> octet2								
octet1 <b>XOR</b> octet2								
<b>NON</b> (octet1)								
<b>NON</b> (octet2)								

## 2 Introduction aux programmes de manipulation d'octet

Dans ce programme, on définira le type OCTET comme une application des caractères *non signés* en définissant un nouveau type grâce à l'instruction :

```
typedef unsigned char OCTET ;
```

## 3 Programme de manipulation d'octet (fichier TP8e2 .cpp)

Écrire un programme principal très simple qui servira à tester les fonctions et procédures qui suivent. Il fera une saisie filtrée d'un entier  $n$  (de type `int`) entre 0 et 255, puis le convertira en OCTET par un transtypage :

```
octet = (OCTET)n ;
```

où `octet` est la variable de type OCTET qui servira par la suite.

### 3.1 Procédure d'affichage d'un OCTET

#### 3.1.1 Opérateurs de décalage

Si `oct1` est l'octet 0000 0001, l'opérateur `<<` permet de décaler ses bits à gauche en faisant entrer des 0 à droite :

```
oct1 << 1    donne    0000 0010 (Attention oct1 n'est pas modifié)
```

```
oct1 << 3    donne    0000 1000
```

etc.

Si `oct2` est l'octet 0101 0011, l'opérateur `>>` permet de décaler ses bits à droite en faisant entrer des 0 à gauche :

```
oct2 >> 1    donne    0010 1001
```

```
oct2 >> 2    donne    0001 0100
```

etc.

#### 3.1.2 Notion de masque

Un masque est un OCTET utilisé de façon particulière.

Soit 0000 0001 un OCTET appelé `mask`. Que vaut l'expression `mask = mask << 7 ;` ?

Combien vaut `mask & oct2 ;` ?

Et si `mask` avait valu `mask = mask << 6 ;`, qu'aurait-on obtenu ?

#### 3.1.3 Procédure `AffOct ( )`

En utilisant les remarques précédentes, imaginer une procédure `AffOct ( )` qui reçoit un octet nommé `oct` et qui, sans rien retourner, affiche les bits de cet octet dans l'ordre de l'écriture « européenne ».

Par exemple, l'octet 255 doit donner sur l'écran 1111 1111 et l'octet 1 doit donner 0000 0001 (et non pas 1000 0000 !).

### 3.2 Fonction de comptage des bits à 1

Écrire une fonction nommée `NbBitsA1()` qui reçoit un OCTET nommé `oct` et qui retourne le nombre de bits à 1 qu'il contient.

Compléter le programme principal pour tester cette fonction.

### 3.3 Procédures de mise à 0 ou à 1 d'un bit d'un OCTET

On veut écrire une procédure `MetBitA0()` qui reçoit un OCTET et un indice entre 0 et 7 ; cette procédure met à 0 le bit de l'octet ayant l'indice reçu en argument et ne retourne rien.

Quel prototype proposez-vous pour cette procédure ? Justifiez votre choix. Réalisez son codage en C.

Sur le même modèle, écrire une procédure `MetBitA1()` qui reçoit un octet et un indice entre 0 et 7, cette procédure mettant à 1 le bit de l'OCTET ayant l'indice reçu en argument.

### 3.4 Procédure d'échange de deux bits dans un OCTET

La procédure `EchangeBits()` reçoit un OCTET et deux indices, elle réalise l'échange dans l'octet des bits ayant les indices reçus en arguments. Elle ne retourne rien.

En choisissant la technique référence ou pointeur, réalisez son codage en C.

### 3.5 Procédure de mélange des bits dans un OCTET

La procédure `MelangeBits()` réalise le mélange des bits d'un octet qu'elle reçoit en argument... et ne retourne rien.

Réaliser son codage à partir des indications suivantes :

- tirez au hasard deux indices *différents* de l'octet ;
- appelez la procédure `EchangeBits()` avec trois arguments : l'octet et ces deux indices.

## 4 Programme de synthèse (fichier TP8e3.cpp)

Écrire un programme de synthèse qui, après avoir choisi un OCTET au hasard (entre 0 et 255) l'affiche, mélange ses bits, le réaffiche, et recommence à chaque appui d'une touche (voir la figure 8). Tapez sur la touche Echap<sup>1</sup> pour arrêter le programme.

```

Tirage aleatoire d'un octet entre 0 et 255 : 35
Son ecriture binaire est : [ 0010 0011 ]

Echange aleatoire de 2 bits : <"Espace" pour continuer, "Echap" pour arreter>

[ 0010 0110 ] <bits 0 et 2>
[ 0100 0110 ] <bits 5 et 6>
[ 0001 0110 ] <bits 4 et 6>
[ 0001 0101 ] <bits 1 et 0>
[ 0010 0101 ] <bits 4 et 5>
[ 0010 0101 ] <bits 7 et 6>
[ 0100 0101 ] <bits 5 et 6>
-

```

FIG. 8 – Mélange des bits d'un octet

<sup>1</sup>Cette touche a pour code Ascii le n° 27. On peut écrire par exemple : `do .. while(getch() != 27) ;`

## CE QU'IL FAUT RETENIR

- ☞ pour représenter un octet, on utilise le type « unsigned char » ;
- ☞ les opérateurs de manipulations de bits, à ne pas confondre avec les opérateurs logiques.

## 5 À préparer pour la prochaine fois

*Tout réviser pour le partiel !*