L3 et Magistère $1^{ère}$ année, Physique Fondamentale, année 2019-2020

Cours de langage C appliqué à la physique

Bartjan van Tent

1. Introduction
2. Bases *Linux* , premier programme C, compilation, exécution
3. Représentation binaire des nombres entiers et réels en mémoire
4. Variables, constantes, types, opérateurs
5. Tests et boucles
6. Entrées, sorties, fichiers, en C++
7. Fonctions
8. Adresses, pointeurs
9. Tableaux dynamiques
10. Générateur aléatoire uniforme sur [0,1], applications
11. Introduction à la programmation de la résolution numérique d'équations différentielles par les méthodes d'Euler et de Runge-Kutta
12. Structures
13. Fichiers sources, compilation séparée, visibilité

Polycopié écrit par François Naulin

(Consulter les mises à jour sur le site : *http ://hebergement.u-psud.fr/mpo-informatique/*)

1. Introduction

1 But du cours et des TD

L'intitulé général du cours et des TD est « Informatique » mais les sujets abordés ne représentent qu'un aspect bien particulier de ce domaine : l'apprentissage du langage C en vue d'application à la physique. On abordera les sujets suivants :

- éléments du langage *C ANSI*
- notions de *Linux* pour utilisateur
- règles générales de programmation, indépendantes du langage utilisé, à observer pour écrire des programmes justes, efficaces et lisibles donc faciles à faire évoluer.

Liste des documents mis à la disposition des étudiants :

- transparents de cours
- polycopié de cours
- polycopié de TD comprenant :
  - énoncés d'exercices
  - résumé des commandes de base de l'éditeur *emacs*
  - résumé des commandes *Linux* de base

Tous les documents sont disponibles sur le site : *http ://hebergement.u-psud.fr/mpo-informatique/*

2 Le système d'exploitation Linux

Le système d'exploitation est *Linux* (*Debian*). Pour l'utilisateur il apparaît comme un langage de commande (complètement indépendant du C) qui permet de faire effectuer à la machine toutes sortes d'opérations : création et manipulation des fichiers, communication avec différents périphériques, entre utilisateurs, avec le réseau *Internet*, installation, création, compilation et exécution de logiciels, etc.
Linux est un système d'exploitation stable, gratuit et libre. Ceux qui, au prix d'un peu de travail, peuvent l'installer sur leur ordinateur personnel, retrouvent un environnement identique à celui du Magistère et de la plupart des laboratoires de recherche et de nombreuses entreprises. Ils ont accès gratuitement à de nombreux logiciels, en particulier scientifiques, de très haute qualité. Son inconvénient est qu'il ne permet pas toujours aisément l'utilisation de certains périphériques ou l'installation de nouveaux logiciels. La documentation sur *Linux* abonde aussi bien dans les librairies que sur Internet. Ce qu'on en apprendra, le strict nécessaire pour nos besoins, n'est qu'une toute petite partie de ses possibilités.
Pour l'écriture des programmes il est proposé d'utiliser l'éditeur [1] *emacs*. D'autres éditeurs pourront être installés si nécessaire.

3 Le langage C

C is not just another programming language, it's the lingua franca of programming and the bedrock of modern computing ; most operating systems, networks, web browsers and many other programming languages such as Python are written in C.

Né au début des années 70, c'est un langage polyvalent et universellement répandu, particulièrement utilisé par les informaticiens.
Ses avantages sont, en particulier les suivants :

- il est à la fois pas trop éloigné du langage de base de l'ordinateur, ce qui permet une communication assez directe et rapide avec lui, et évolué, ce qui permet une communication pas trop laborieuse avec le cerveau humain
- il est aisé de faire interagir les programmes C avec le système d'exploitation
- il est la base du C++
- il existe des compilateurs [2] gratuits d'excellente qualité, ainsi qu'une documentation abondante, gratuite ou bon marché.

1. Un éditeur est un logiciel permettant d'écrire du texte brut (par opposition au texte mis en forme par un traitement de texte) dans un fichier.
2. Le compilateur est le programme qui traduit le C en instructions exécutables par la machine.

Exemple des temps d'exécution (en ms) d'un programme test (tri) selon le langage :

Langage	C	Java	PHP	Python
Temps d'exécution	2.7	47.8	89.2	92.

Ses inconvénients :

— pour le calcul scientifique il n'a pas l'élégance et la clarté du Fortran (mais ce dernier n'a pas la même polyvalence et est principalement cantonné aux laboratoires de recherche alors que le C est largement utilisé dans les entreprises et, d'autre part, le C++ permet de compenser les lacunes du C)

— il se prête bien à l'écriture de programmes remarquablement obscurs. Il existe d'ailleurs un concours international annuel du programme le plus incompréhensible.

C'est donc le C qui est enseigné, en vue du C++ étudié en seconde année de Magistère. Quelques éléments mineurs de C++ [3] sont cependant introduits dès la première année, quand ils permettent de faire plus simplement que leurs équivalents en C.

Remarque

Dans le domaine scientifique *Maple* et *Mathematica* sont très intéressants pour le calcul formel mais trop lents pour les calculs numériques.

Ce cours s'adresse non seulement à ceux qui n'ont jamais programmé en C, mais aussi à ceux qui n'ont jamais programmé du tout. Seuls des rudiments y sont présentés. Des notions importantes sont à peine abordées (préprocesseur, fichiers, chaînes de caractères, structures). De nombreuses subtilités ne sont pas mentionnées. Cependant ce qui est présenté permet déjà de faire des calculs intéressants, en particulier en physique. De plus ce cours voudrait être une base suffisante à partir de laquelle il est aisé d'étendre ses connaissances à l'aide de documents écrits.

4 Liste de livres de référence pour Linux et le C

En principe l'essentiel de ce qui est au programme doit se trouver dans le cours et il n'est pas nécessaire d'acheter un livre. Cependant, il peut être très utile d'en consulter un pour vérifier une syntaxe ou un point délicat, obtenir un complément d'explication ou étudier des éléments du C qui ne sont que peu ou pas abordés dans le cours.

Pour débuter :

— Le livre du C premier langage, Claude Delannoy, (s'adresse en particulier à des lecteurs qui ne connaissent aucun langage de programmation mais peut convenir aussi aux autres pour démarrer le C)

— Programmer en langage C, Claude Delannoy, (plus complet que le précédent, reste abordable pour un lecteur commençant l'étude du C)

Pour approfondir les points délicats ou les notions complexes, des livres tendant vers l'exhaustivité mais difficiles pour un débutant :

— La référence du C norme ANSI/ISO, Claude Delannoy

— Le langage C Norme ANSI, Brian W. Kernighan et Denis M. Ritchie, (écrit par les fondateurs du C)

— Le langage C ANSI, Philippe Drix, Dunod

Il existe aussi de très bons sites pour le C sur Internet, voir en particulier : *http ://cpp.developpez.com/cours/cpp/* et, plus généralement *http ://www.developpez.com/*

Pour *Linux* le cours oral et le polycopié suffisent largement. Pour ceux qui veulent en apprendre plus, plusieurs livres très utiles sont disponibles à la bibliothèque universitaire, tel :

— Linux in a Nutshell, J.P. Hekman, etc. O'REILLY

5 Les logiciels mis à disposition des étudiants

— compilateurs C, C++

— *Python*, *Matplotlib* (langage interprété, graphisme)

— *Gnuplot*, *root* (tracé de courbes et surfaces, traitement de données)

3. Signalés explicitement comme tels.

— bibliothèque mathématique *gsl*
— *Xfig*, *Inkscape* (dessin vectoriel)
— *Latex* (traitement de texte, en particulier scientifique)
— *OpenOffice* (analogue à *Office* de *Microsoft*)
— *Gimp* (traitement d'images)
— *OpenGL* (animation)

et tous les logiciels courants d'une distribution *Linux*.

2. Bases Linux, premier programme C, compilation, exécution

1 Bases de *Linux* pour l'utilisateur

1.1 Commandes, ligne de commande

Pour utiliser un ordinateur fonctionnant avec *Linux* il faut d'abord fournir un « identifiant » (c'est à dire un nom d'utilisateur) et un mot de passe[4]. Dans la suite on prend l'exemple d'un utilisateur nommé « averell » et d'un ordinateur nommé « vega » .
Ensuite des commandes peuvent être transmises de deux façons :

1. par des icônes, des menus et des raccourcis clavier (comme dans *Microsoft Windows*)
2. par l'écriture de ces commandes dans une fenêtre particulière nommée « Terminal » . Cette écriture se fait à l'aide du clavier mais aussi de la souris et de quelques procédés qui permettent de réduire au minimum les caractères à taper.

C'est la seconde méthode qui est décrite ici.
Le système indique qu'il est prêt à recevoir des commandes dans la fenêtre Terminal en inscrivant à l'écran, dans cette fenêtre, pour l'exemple envisagé ici, la suite de caractères :

[averell@vega ~]$

qui s'appelle l'« invite » (ou « prompt »). Pour abréger on l'écrira parfois simplement $ dans la suite. L'espace vierge situé à la suite de cette invite, sur la même ligne, s'appelle la « ligne de commande » , c'est là que l'utilisateur écrit ses commandes.

Remarque :

Dans l'écriture d'une commande *Linux*, minuscules et majuscules n'ont pas la même signification.

1.2 Fichiers, répertoires

Les fichiers sont des ensembles d'informations écrites sous forme de 0 et de 1 sur des supports tels que mémoire vive, disques durs, clés USB, bandes magnétiques, cartes mémoire, CD, DVD, etc. A chaque utilisateur est attribuée une zone personnelle sur le disque dur de l'ordinateur, dans laquelle il crée, modifie et conserve ses fichiers.
Ces fichiers peuvent avoir des contenus très divers :

- texte brut directement lisible (fichier dit « texte ») qui peut représenter par exemple :
  — un programme écrit en C ou en tout autre langage
  — des données sous forme de chiffres ou de caractères alphanumériques
  — un texte ordinaire non mis en forme
- informations écrites dans un codage particulier, lisibles seulement par l'intermédiaire d'un logiciel et représentant par exemple :
  — un programme C (ou autre langage) sous forme compilée
  — un texte mis en forme par un traitement de texte
  — des figures, images, sons etc.

Les fichiers textes peuvent être créés et modifiés par l'utilisateur, soit directement à l'aide d'un « éditeur » (ensemble de commandes permettant d'écrire, depuis le clavier, des caractères dans une fenêtre de l'écran[5] et de sauvegarder ce qui est écrit dans cette fenêtre dans un fichier), soit indirectement par l'intermédiaire d'un processus quelconque effectué par la machine (par exemple fichier contenant les résultats d'un calcul fait par un programme C).

Conseil pratique :

Avoir un fichier (unique) dans lequel on note toutes les informations utiles que l'on écrit habituellement sur de petits bouts de papier.

1.2.1 Répertoires, arborescence

De même que, dans la vie courante, il est plus pratique de ranger ses papiers dans plusieurs dossiers, plutôt que dans un seul où tout se mélange, les fichiers sont classés dans des ensembles nommés *répertoires*[6].

4. En général plusieurs utilisateurs sont enregistrés sur un ordinateur donné et sont donc susceptibles de l'utiliser.
5. Différente de la fenêtre Terminal.
6. Éventuellement imbriqués les uns dans les autres.

L'utilisateur peut créer à sa guise, une structure en arbre constituée de répertoires et de fichiers auxquels il attribue des noms de son choix. La figure suivante donne un exemple d'une telle structure :

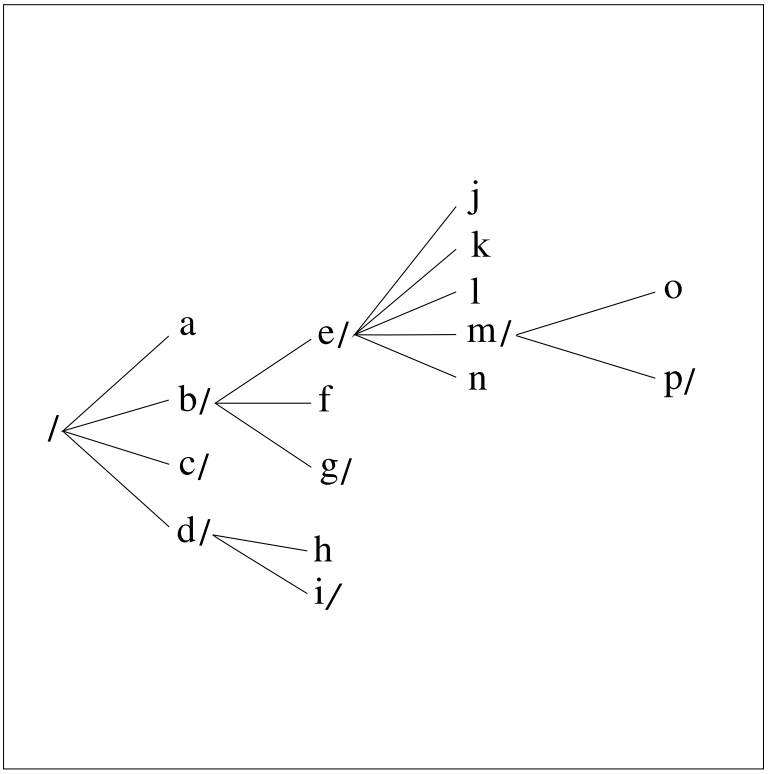


FIGURE 1 – Exemple de structure arborescente de répertoires et de fichiers

Dans cet exemple on a nommé les répertoires et les fichiers à l'aide de simples lettres, pour simplifier la description. En général on leur donne des noms plus explicites pour mieux s'y retrouver (voir exemple ci-dessous). Sur la figure les répertoires se distinguent des fichiers par le fait que leur nom est suivi d'un « slash » : « / » . Par exemple *b* est un répertoire contenant deux répertoires (*e* et *g*) et un fichier (*f*). *e*, à son tour, contient un répertoire et quatre fichiers.

1.2.2 Répertoires particuliers

« répertoire initial »
: répertoire personnel de l'utilisateur dans lequel il se trouve initialement placé après s'être identifié sur l'ordinateur. Il porte le nom de l'utilisateur (par exemple *averell*) et il peut être désigné de façon générique par un « tilde » : « ∼ » . C'est dans son répertoire initial qu'un utilisateur crée tous ses répertoires et ses fichiers.

« répertoire courant »
: répertoire dans lequel se trouve l'utilisateur au moment où il écrit une commande. Son nom est inscrit à la fin de l'invite, il peut être désigné de façon générique par un point : « . » .

« répertoire père »
: répertoire immédiatement ascendant du répertoire courant, il peut être désigné de façon générique par deux points : « .. » .

« répertoire racine »
: répertoire le plus en amont, le père de tous, il est désigné par un slash : « / »

1.2.3 Chemin d'accès

Le « chemin d'accès » d'un répertoire ou d'un fichier est la suite des répertoires qui permettent d'arriver jusqu'à lui depuis le répertoire racine inclus. Dans l'exemple de la figure 1, le chemin d'accès du répertoire *p* est */b/e/m*.

Le nom complet d'un répertoire ou d'un fichier est son chemin d'accès suivi de son nom local. Le nom complet du fichier o est $/b/e/m/o$.
Pour désigner un fichier à partir d'un répertoire quelconque de l'arbre il faut, en principe, donner son nom complet, en particulier parce que plusieurs répertoires ou fichiers de pères différents peuvent porter le même nom (déconseillé), mais des simplifications apparaissent souvent :

— si le répertoire ou fichier que l'on veut désigner est un descendant du répertoire courant il suffit de donner le chemin à partir du fils du répertoire courant. Par exemple, pour désigner o :

quand on est dans e : m/o
quand on est dans b : $e/m/o$

— si le répertoire ou fichier que l'on veut désigner n'est pas un descendant du répertoire courant on peut quand même simplifier en donnant le nom complet sous forme relative par rapport au répertoire courant. En effet, comme « .. » désigne de façon abrégée le répertoire père du répertoire courant, pour désigner m en étant dans g on peut écrire $../g/e/m$.

La désignation des fichiers et des répertoires est également facilitée par l'utilisation des caractères génériques et le mécanisme de « complétion » des noms de fichiers (voir polycopié *Linux*), ainsi que par l'utilisation du copier-coller avec la souris (voir TD). Il n'est jamais nécessaire d'écrire complètement le nom d'un fichier pour le désigner, sauf, évidemment, au moment de sa création.

1.2.4 Quelques commandes *Linux* pour manipuler les fichiers et les répertoires

pwd : affichage du nom de répertoire courant

Ainsi, si le répertoire courant est e, et en notant l'invite par $\$$:

$\$\ pwd$

donne le résultat :

$/b/e$

cd : déplacement dans les répertoires

Si l'utilisateur se trouve dans le répertoire b et écrit la commande :

$\$\ cd\ e$

il passe de b à e. Le répertoire courant qui était b devient e. Si l'utilisateur se trouve dans le répertoire g et écrit la commande :

$\$\ cd\ ../e/m$

le répertoire courant qui était g devient m.
La figure suivante illustre quelques cas possibles.

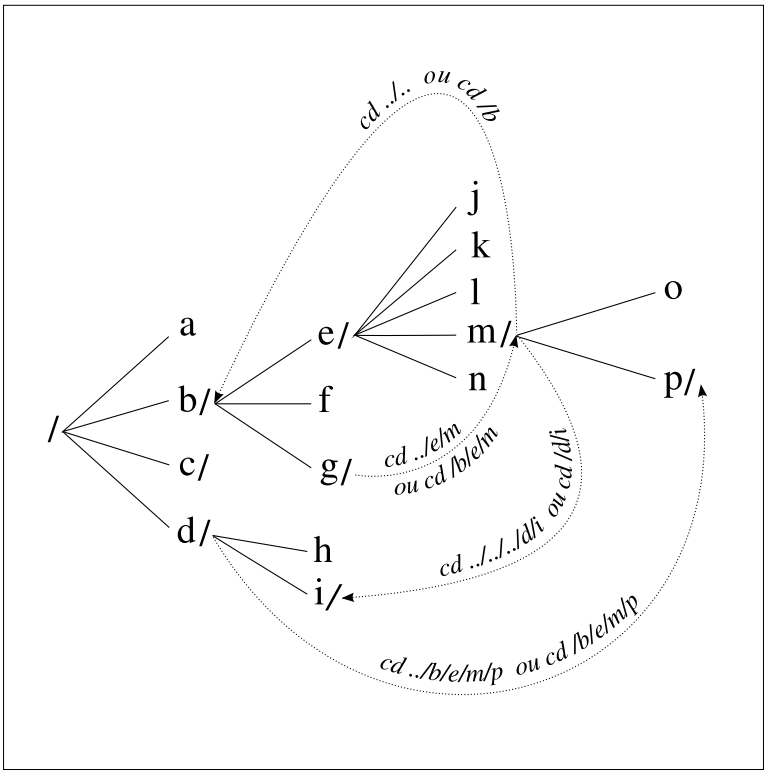


FIGURE 2 – Exemples de passage d'un répertoire à un autre, attention au sens des flèches

Exercice :

Ecrire les commandes permettant de faire les déplacements inverses de ceux indiqués par les quatre flèches de la figure 2.

La commande *cd* employée seule (sans « argument ») fait revenir au répertoire initial. Elle est donc équivalente à *cd* ∼.

ls : liste des fichiers du répertoire courant

Si on est dans le répertoire *e* :

$ ls

donne le résultat :

j k l m n

A l'écran les noms des répertoires apparaissent avec une couleur, ceux des fichiers avec une autre. Par exemple *j k l n* seront en noir, *m* en bleu. Si on veut que les noms de répertoires soient suivis d'un slash comme sur la figure 1, il faut écrire la commande *ls* avec l'option *F*[7].

$ ls -F

donne le résultat :

j k l m/ n

cp : copier un fichier

$ cp x y

copie le contenu du fichier *x* dans le fichier *y*. Si *y* n'existait pas il est créé, s'il existait son contenu antérieur est remplacé par celui de *x*. Dans ce dernier cas, si *cp* est employé avec l'option *i*, par précaution le système demande confirmation avant de remplacer le contenu de *y* par celui de *x*.

mv : changer le nom d'un fichier ou le déplacer

7. L'option d'une commande est précédée d'un tiret : « - » .

$ mv x y

Si x est un fichier et y n'existe pas, ou est aussi un fichier, le nom de x est changé en y, l'ancien x et l'ancien y s'il existait, disparaissent. Avec l'option i, une confirmation est demandée si y existait.
Si x est un fichier ou un répertoire et y un répertoire, x est déplacé du répertoire courant dans le répertoire y.
Si x est un répertoire et y n'existe pas, le nom de x est changé en y.

rm : supprimer un fichier

$ rm x

Avec l'option i, une confirmation de la suppression est demandée.

del : supprimer un fichier

$ del x

C'est une abréviation pour *rm -i*, ce n'est pas une vraie commande *Linux* , mais une commande créée pour l'environnement du Magistère[8]. Il faut l'utiliser à la place de *rm* qui est trop risquée.

touch : créer un fichier vide

$ touch x

crée un fichier vide de nom x.

mkdir : créer un répertoire

$ mkdir x

crée un répertoire vide de nom x dans le répertoire courant

rmdir : supprimer un répertoire

$ rmdir x

Supprime le répertoire x à condition qu'il soit vide.

cp -a : copier récursivement un répertoire

$ cp -a x y

si x est un répertoire et y n'existe pas, crée une copie de x de nom y récursivement, c'est à dire contenant tous les répertoires descendant de x.

kompare : compare deux fichiers

$ kompare x y

compare les deux fichiers x et y en les affichant côte à côte et en faisant apparaître les différences dans des plages colorées.

rename : permet de renommer des ensembles de fichiers

*$ rename 's/\.abc$/def/' *.abc*

change, pour tous les fichiers d'extension *abc*, cette extension en *def*. Mais si le fichier contient aussi la chaîne *abc* dans son nom, c'est à dire avant l'extension, cette chaîne ne sera pas modifiée.

locate : permet de savoir où se trouve un fichier dans l'arborescence

$ locate xyz

affiche le nom complet d'un fichier dont le nom contient la chaîne *xyz*. *locate* ne recherche pas directement dans l'arborescence actuelle mais dans une base de données qui n'est pas forcément à jour. Pour la mettre à jour il faut utiliser la commande *updatedb*.

find : permet de rechercher des groupes particuliers de fichiers

$ find lulu -name ''xyz'*' -print*

affiche le nom complet de tous les fichiers situés sous le répertoire *lulu* dans l'arborescence et qui contiennent la chaîne *xyz* dans leur nom. On peut imposer toutes sortes de conditions supplémentaires sur les propriétés des fichiers recherchés.

grep : permet de chercher une chaîne de caractères dans un ou un ensemble de fichiers textes dont les noms sont connus

8. Tout utilisateur peut créer ses commandes personnelles.

$ grep toto fifi

affiche toutes les lignes du fichier *fifi* qui contiennent la chaîne *toto*

find et ***grep*** combinés : permet de chercher une chaîne de caractères dans un ou un ensemble de fichiers textes dont les noms sont inconnus

find lulu -print | xargs grep Jules

affiche le nom complet de tous les fichiers situés sous le répertoire *lulu* dans l'arborescence et qui contiennent la chaîne de caractères *Jules*. La ligne du fichier qui contient la chaîne de caractères *Jules* est également affichée.

man : afficher à l'écran le mode d'emploi d'une commande

$ man ls

explique le rôle et les différentes options de la commande *ls*.

which : indique où se trouve une commande dans l'arborescence

$ which ccc

donne le chemin d'accès de la commande *ccc*.

Exercice :

Créer une arborescence identique à celle de la figure 1. On peut créer plusieurs répertoires à la fois par *mkdir x y z* et plusieurs fichiers à la fois par *touch u v w*, par exemple.

Toutes ces commandes *Linux* et quelques autres sont également décrites dans le polycopié *Linux*, chapitre I, inclus dans le polycopié de TD.

1.2.5 Vue d'ensemble de l'arborescence de fichiers d'un système *Linux*

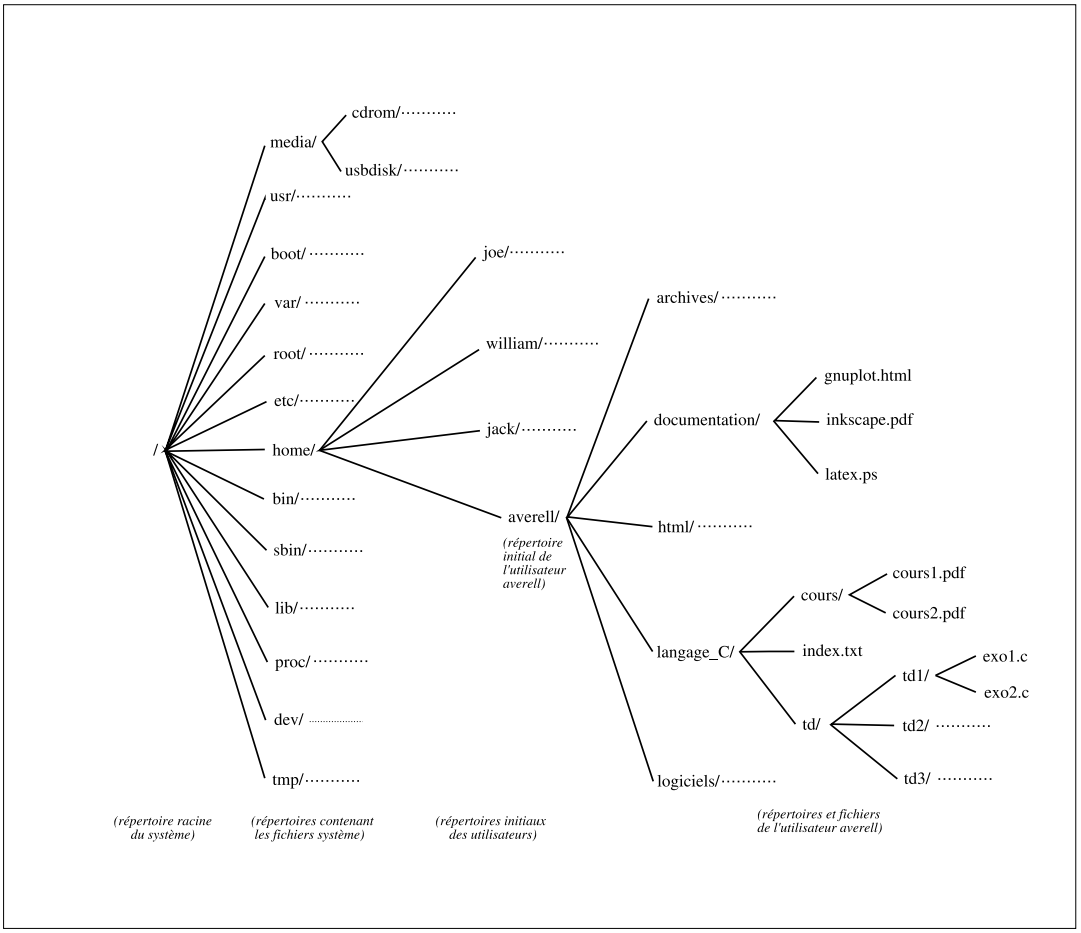


FIGURE 3 – Exemple d'arborescence de fichiers d'un système *Linux*

Les répertoires initiaux des utilisateurs[9] sont tous situés dans un répertoire nommé *home*. Ce répertoire fait partie d'une arborescence plus vaste constituée des fichiers système. Un utilisateur quelconque peut, à l'aide de la commande *cd*, se déplacer en dehors de ses répertoires personnels, à condition d'avoir les droits d'accès. Ainsi averell peut, en faisant, à partir de son répertoire initial, *cd ../joe*, aller dans les répertoires de l'utilisateur joe et y lire, copier, modifier et détruire des fichiers, à condition que joe lui en ait donné l'autorisation. De même un utilisateur quelconque peut lire, mais pas modifier, certains fichiers système. La façon de fixer les droits d'accès est expliquée dans un chapitre consacré à *Linux* .

1.3 L'éditeur

L'éditeur est l'ensemble des commandes qui permettent d'écrire dans un fichier à partir du clavier.
On dispose de deux éditeurs : *vi* et *emacs*, par ordre de sophistication croissante. On utilise *emacs* dont un mode d'emploi simplifié est donné dans le polycopié de TD.

2 Premier programme en C

Le programme le plus simple qu'on puisse écrire en C est :

```
int main() {
```

9. Ici il y a quatre utilisateurs : joe, william, jack et averell.

```
  return 0;
}
```

Il ne fait presque rien. Il mérite cependant d'être mentionné car tout programme est écrit à partir de ce point de départ.

`int main()`

indique le début du programme principal qui est celui dont le déroulement dirige tout le reste. Les accolades qui suivent contiennent l'unique instruction du programme principal :

`return 0;`

qui retourne la valeur 0 à la commande *Linux* qui a fait exécuter ce programme. Cette valeur 0 signifie que le programme s'est exécuté normalement.
Dans cet exemple il n'y a rien d'autre que le programme principal.

Considérons le programme :

```
#include<iostream>
using namespace std;
int main() {
  double x, y, z;
  x = 2.; y = 3.; z = x/y;
  cout << "Resultat : " << z << endl;
  return 0;
}
```

supposé écrit dans un fichier de nom, par exemple, *monprog.cpp* (on adopte comme convention obligatoire de suffixer les fichiers qui contiennent du C/C++ par *.cpp*).

Commentaire détaillé :

`#include<iostream>`
`using namespace std;`

indiquent qu'on utilise la bibliothèque d'entrée-sortie pour pouvoir écrire le résultat

`double x, y, z;`

indique qu'on déclare trois variables réelles nommées `x, y, z`

`x = 2.; y = 3.; z = x/y;`

attribue des valeurs à `x` et `y`, et attribue la valeur du quotient `x/y` à `z`

`cout << "Résultat : " << z << endl;`

fait imprimer à l'écran le mot *Résultat :* suivi de la valeur de `z`.
`endl` fait aller à la ligne après ce qui précède.

Une fois que ce programme est écrit il faut procéder à deux opérations pour obtenir son résultat :

1. la « compilation »
2. l' « exécution » .

2.1 Compilation

La compilation est la traduction des instructions du C en un langage plus adapté au fonctionnement de la machine que l'on utilise. Chaque instruction du programme C, assez proche de la pensée humaine, est développée en une suite d'instructions plus élémentaires dont l'écriture directe serait laborieuse. On peut comparer la compilation à ce qui se passe lorsqu'on fait une addition avec un boulier chinois. Le programme d'origine serait, par exemple, « ajouter 5 et 7 » et le programme compilé la description de la séquence d'opérations élémentaires à effectuer avec le boulier pour parvenir à ce résultat. Durant la compilation la machine analyse le programme sans exécuter les instructions. Elle évalue la taille des emplacements de mémoire qui sont nécessaires. Si elle détecte une incohérence ou un non respect des règles elle le signale : on dit qu'il y a une erreur à la compilation. Si elle ne dit rien ceci ne veut pas dire qu'il n'y a pas d'erreur : ce qu'on a écrit peut avoir un sens en C mais différent de celui auquel on pense et les ennuis viendront plus tard. Le résultat de la compilation est écrit dans un nouveau fichier dit « exécutable » qui, dans notre cas, porte le nom *a.out* quel que soit le nom du fichier dit « source » qui contient le C. En accord avec ce qui précède, le fichier exécutable est beaucoup plus volumineux que le fichier source. De plus, alors que le fichier source est indépendant

de la machine utilisé, l'exécutable lui, ainsi que le compilateur qui le fabrique, est complètement lié à cette machine. On peut utiliser *monprog.cpp* tel quel sur n'importe quelle machine. Mais le *a.out* obtenu avec le compilateur d'une machine X ne peut, en général, fonctionner sur une machine Y. *monprog.cpp* doit être recompilé avec le compilateur de la machine Y. La compilation étant une traduction dans un langage plus proche de la machine, il n'est pas étonnant que le résultat dépende fortement de l'architecture de cette dernière. Pour l'utilisateur la compilation est une étape qui ne paraît pas très importante car pratiquement tout est caché mais c'est en réalité une opération extrêmement complexe.
En pratique, pour compiler le programme écrit dans le fichier *monprog.cpp*, il faut écrire la commande :

g++ -lm -Wall monprog.cpp

On peut vérifier la présence dans le répertoire courant du fichier nouvellement créé *a.out*, à l'aide de la commande *ls*.

2.2 Exécution

C'est l'étape durant laquelle la machine se met à exécuter une à une les instructions, dans l'ordre où elles lui sont indiquées et à l'issue de laquelle deux cas se présentent :

- l'exécution s'arrête en cours de route, en émettant un commentaire lapidaire. Il faut alors comprendre ce qui ne va pas : regarder jusqu'où le programme s'est déroulé correctement pour repérer l'endroit où survient le problème, au besoin en ajoutant des impressions régulièrement réparties. Relire aussi le fichier source en vérifiant les points qui sont le plus souvent à l'origine des erreurs et dont une liste est donnée à l'annexe **Erreurs les plus fréquentes**.
- l'exécution va à son terme et fournit des résultats. Bien sûr ce n'est pas parce que la machine fournit un résultat qu'il est juste et c'est alors à l'utilisateur d'utiliser toute sa perspicacité pour en décider. Ce qui est sûr c'est que la machine a fait ce qu'on lui a dit de faire, qui n'est pas forcément ce qu'on croit lui avoir dit. Elle ne se trompe pas aléatoirement, même rarement, ce qui est assez remarquable, vu le nombre gigantesque d'opérations élémentaires que nécessite l'exécution du moindre programme.

Remarque

Il y a deux sortes de langage :

- les langages compilés : toutes les instructions doivent être fournies à l'ordinateur avant qu'il ne commence à exécuter (exemples : C, Fortran, Pascal)
- les langages interprétés : l'exécution se fait immmédiatement après chaque instruction fournie (exemples : shell Linux, Maple, Mathematica, Python)

En pratique, pour faire exécuter le programme écrit dans le fichier *monprog.cpp*, il faut écrire la commande :

./a.out

On obtient alors sur l'écran :

Resultat : 0.666667

En réalité, il existe une abréviation[10] nommée *ccc*, créée pour l'environnement du Magistère, qui permet de compiler et d'exécuter en une seule commande :

ccc monprog.cpp

fait apparaître directement le résultat à l'écran. C'est cette commande *ccc* qu'on utilisera, sauf cas particulier[11]

2.3 Indications complémentaires sur les éléments d'un programme

2.3.1 Marqueurs, directives, instructions de déclaration, instructions exécutables

Dans le programme *monprog.cpp* précédent on peut distinguer quatre types d'éléments :

- `int main()`, `{` et `}` sont de simples marqueurs, ils indiquent où commence et finit le programme principal. On dit que `int main()` est un « en-tête » et que `{` et `}` délimitent un « bloc » .

10. Plutôt qu'abréviation on emploie le terme « fichier de commande » .
11. Par exemple si *monprog.cpp* n'a pas été modifié depuis la dernière exécution, l'utilisation de *ccc* recompile inutilement. Vu que *ccc* supprime le fichier *a.out* après l'avoir exécuté, il est mieux d'utiliser explicitement la commande *g++* donnée ci-dessus pour compiler le programme une fois et puis l'exécuter autant de fois qu'on veut avec *./a.out*, si l'on compte exécuter le même programme plusieurs fois.

— `#include<iostream>` est une « directive » . Elle doit être placée avant la partie du fichier à laquelle elle doit s'appliquer (ici avant le programme principal). Elle doit être écrite en commençant au début de la ligne et il ne peut y en avoir deux par ligne. Elle est prise en compte lors de la compilation.
— `double x, y, z;` est une instruction de déclaration. Elle est prise en compte lors de la compilation. Les déclarations situées à l'intérieur du bloc contenant les instructions du programme principal doivent être placées au début de ce bloc.
— `x = 2.; y = 3.; z = x/y; cout << "Résultat : " << z << endl; return 0;` sont des instructions exécutables, prises en compte seulement lors de l'exécution.

Une instruction doit être terminée par un point virgule. Elle peut s'étendre sur plusieurs lignes, elle n'est pas terminée tant que le point virgule n'est pas rencontré. Inversement on peut regrouper plusieurs instructions sur la même ligne, à condition de les séparer par des points virgule. Il peut être intéressant de le faire, pour éviter que le fichier ne s'étire trop en longueur. Mais pour assurer la lisibilité du programme il ne faut regrouper que des instructions ayant entre elles un certain rapport de sens (par exemple l'attribution de valeur pour `x = 2.; y = 3.; z = x/y;` dans l'exemple).

2.3.2 Commentaires

Il existe trois façons d'insérer des commentaires dans le programme, c'est à dire du texte qui n'est pas pris en compte par l'ordinateur mais sert à l'humain qui écrit ou utilise le programme. Ce peut être des commentaires au sens premier du mot ou des portions de programme que l'utilisateur souhaite neutraliser provisoirement.

1. Deux slash contigüs transforment en commentaire tout ce qui les suit sur leur ligne :

   ```
   x = 2.5; // x est exprimé en mètres
   ```

   Ne permet de commenter qu'une ligne à la fois, pratique pour mettre en commentaire de très petites portions de programme.

2. Toute ce qui est compris entre `/*` et `*/` est transformé en commentaire :

   ```
   /*
   a = b*c;
   f = sin(u);
   */
   x = 2.5;
   ```

   Les instructions `a = b*c; f = sin(u);` sont neutralisées. Permet de commenter autant de lignes que l'on veut d'un seul coup mais on ne peut pas imbriquer :

   ```
   /*
   ...
   /*
   ...
   */
   ...
   */
   ```

   ne fonctionne pas, ce qui est gênant pour des programmes un peu longs.

3. Toute ce qui est compris entre `#if 0` et `#endif` est transformé en commentaire (existe uniquement en C++) :

   ```
   #if 0
   a = b*c;
   f = sin(u);
   #endif
   x = 2.5;
   ```

   Même utilisation que `/* ... */` [12], mais on peut imbriquer.

2.3.3 Distinction minuscules-majuscules

En C il y a, comme dans *Linux* , distinction entre les minuscules et les majuscules.

12. Sauf que `#if 0` et `#endif` doivent se trouver chacun seuls sur une ligne, ce qui n'est pas nécessaire pour `/*` et `*/`.

3 Conseils de programmation

3.1 Ecriture

Il faut avoir défini une méthode, c'est à dire un algorithme complet, pour parvenir au but recherché avant de commencer à écrire les instructions (cet algorithme est généralement largement ou totalement indépendant du langage utilisé, C, Pascal, Fortran, Maple, etc. pour la programmation proprement dite). Au fur et à mesure de l'écriture, raisonner rigoureusement pour savoir si le programme va bien faire exactement ce qu'on veut en le faisant exécuter mentalement, comme si on était l'ordinateur. Il ne faut surtout pas écrire des instructions qui doivent approximativement faire le travail en se disant qu'on corrigera ensuite selon les diagnostics fournis par le compilateur et les résultats obtenus. En effet les diagnostics ne sont pas toujours clairs, les comprendre peut être laborieux. Ce qui est plus grave, une erreur du programmeur peut avoir, pour le compilateur, un sens, mais tout a fait différent du bon. Il n'y a évidemment pas, alors, de diagnostic et les résultats sont faux. Si l'utilisateur a un moyen de les estimer il s'aperçoit qu'il y a une erreur, sinon celle-ci passe inaperçue jusqu'à ce qu'un calcul indépendant soit fait, que le pont s'écroule ou que la fusée s'écrase.
Enfin il faut séparer au maximum les tâches indépendantes c'est à dire rendre le programme le plus modulaire possible. Le C n'exige rien de ce point de vue et c'est à l'utilisateur de bien structurer ses programmes.

3.2 Relecture

Le plus difficile n'est pas d'écrire un programme, mais de le relire après un certain temps pour comprendre ce qu'il fait et comment il le fait. Même pour un programme que l'on a écrit soi-même et dont on connaît exactement la finalité, après quelques mois il peut être très long de comprendre l'algorithme employé et sa programmation à partir de la seule lecture des instructions pour, par exemple, modifier ou développer sans faire d'erreurs.

Il faut, en tête du fichier, mettre un commentaire expliquant quelle est la tâche effectuée par le programme et suivant quel algorithme. S'il n'est pas possible d'écrire ceci en commentaire il faut citer une référence.

Il faut programmer de façon claire et naturelle en évitant les astuces inutiles, ne pas chercher à trop condenser sauf si cela apporte réellement un gain et dans ce cas mettre des commentaires. Inversement il faut bannir les instructions et les commentaires inutiles. Cela est encore plus vrai quand le programme doit être utilisé et modifié par d'autres.

4 Lexique Français-Anglais

identifiant = login name
mot de passe = password
répertoire initial = home directory
répertoire courant = working directory
répertoire racine = root directory
fichier de commande = script

3. Représentation des nombres entiers et réels en binaire en mémoire

1 Nombres entiers

1.1 Représentation binaire

Tout entier positif n peut s'écrire sous la forme :

$$n = c_q 2^q + c_{q-1} 2^{q-1} + ... + c_k 2^k + ... + c_1 2^1 + c_0 2^0$$

c_q=1 (q est le plus grand entier tel que $n/2^q \neq 0$ (division « entière » ou « euclidienne »)) et les autres c_k valent 0 ou 1. La suite :

$$c_q c_{q-1} ... c_1 c_0$$

constitue la représentation binaire de n.
Pour calculer les c_k on divise successivement n par 2 et à chaque fois on garde le reste, qui vaut 0 ou 1. La suite des restes constitue la liste des c_k par ordre de poids croissant : $c_0 ... c_q$.
Comme en notation décimale on écrit les forts poids à gauche.
Exemple : 29
suite des restes : 1 0 1 1 1 donc dans le bon ordre : 11101

1.2 Ecriture des entiers en binaire en mémoire

Les PC utilisés en TD codent les entiers (du type `int` que nous utiliserons principalement) dans des mots de 4 octets (=32 bits) dont un est réservé pour le signe (par exemple le plus à gauche, figure 1).

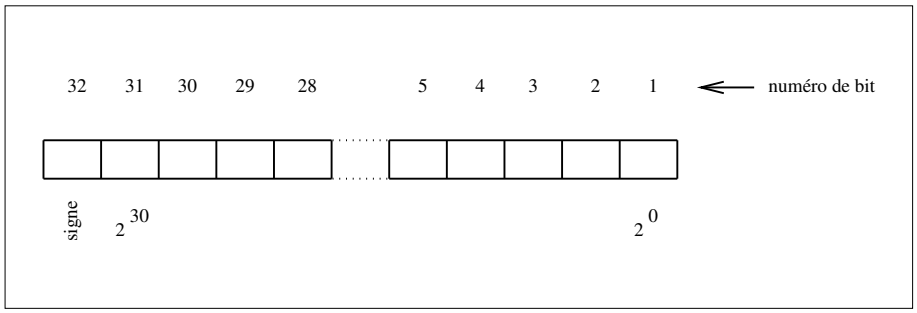


FIGURE 4 –

S'il s'agit d'un entier positif le bit de signe vaut 0 et l'entier lui-même est codé sur les 31 bits restants, on peut donc écrire tous les entiers de 0 à n_{max}=2^{31}-1=2 147 483 647 (ordre de grandeur 2 milliards).
S'il s'agit d'un entier négatif le bit de signe vaut 1. Dans les 31 bits restants on ne code pas la valeur absolue du nombre mais $n + 2^{31} = 2^{31} - |n|$ (cela simplifie l'exécution des opérations).

Exemple

codage de -1
le bit de signe vaut 1
sur les 31 bits on va coder $2^{31} - 1$=1111....1111 (31 fois 1)
donc au total sur les 32 bits des 1 partout

En pratique pour obtenir la représentation binaire de $-n$, n étant un entier positif, on soustrait, en binaire sur 32 bits n de 0 en laissant tomber la dernière retenue.
L'entier négatif de plus grande valeur absolue sera obtenu pour $2^{31} - |n| = 0$ donc n_{min}=-2^{31}= -2 147 483 648 (on gagne la place du 0).
Si lors d'une opération le résultat excède l'intervalle $[n_{min}, n_{max}]$, il n'y a aucun message d'erreur et le résultat est

totalement aberrant (exemple : 2 147 483 647+1=-2 147 483 648).

Exemple :
On calcule les factorielles successives à partir de 1. A partir d'un certain rang on va dépasser l'entier maximum représentable. Pour savoir à quel rang cela se produit, à chaque fois qu'on a calculé $n!$ on divise le résultat par n, c'est à dire qu'on calcule $n!/n$. Si on retrouve $(n-1)!$ c'est que la valeur de $n!$ calculée était correcte, sinon c'est qu'elle était fausse. On obtient :

1 !=1	1 !/1=1
2 !=2	2 !/2=1
3 !=6	3 !/3=2
4 !=24	4 !/4=6
5 !=120	5 !/5=24
6 !=720	6 !/6=120
7 !=5040	7 !/7=720
8 !=40320	8 !/8=5040
9 !=362880	9 !/9=40320
10 !=3628800	10 !/10=362880
11 !=39916800	11 !/11=3628800
12 !=479001600	12 !/12=39916800
13 !=1932053504	13 !/13=148619500
14 !=1278945280	14 !/14=91353234
15 !=2004310016	15 !/15=133620667

On constate que le calcul devient faux à partir de 13 inclus.
A part cette limitation en grandeur, les opérations arithmétiques entre entiers sont exactes (en n'oubliant pas que la division est la division euclidienne : partie entière du résultat de la division exacte, exemple, 2/3=0 et 5/4=1).
Sur les PC ayant un processeur 64 bits il est possible d'utiliser des entiers de type `long int` écrits sur 8 octets (64 bits) donc compris entre -2^{63}=-9 223 372 036 854 775 808 et 2^{63}-1=9 223 372 036 854 775 807 (ordre de grandeur 10 milliards de milliards).

2 Nombres réels

2.1 Représentation binaire

Tout réel positif r peut s'écrire sous la forme :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + c_1 2^1 + c_0 2^0 + c_{-1}2^{-1} + c_{-2}2^{-2} + ... + c_k 2^k + ...$$

avec $c_q = 1$ et les c_k valant 0 ou 1 (q peut être négatif). La suite :

$$c_q c_{q-1} ... c_1 c_0 . c_{-1} ... c_k ...$$

constitue la représentation binaire de r, en général infinie du côté des k négatifs (par convention on met un point entre c_0 et c_{-1}).
Les c_k ne sont jamais tous égaux à 1 à partir d'un certain rang k_0 en direction des k négatifs. En effet on aurait alors :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + c_{k_0+1}2^{k_0+1} + c_{k_0}2^{k_0} + c_{k_0-1}2^{k_0-1} + ... \quad \text{avec } c_{k_0+1} = 0 \text{ et } c_k = 1 \text{ pour } k \leq k_0$$

soit :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + 0 \times 2^{k_0+1} + 2^{k_0} + 2^{k_0-1} + ...$$

Mais, comme on a :

$$2^{k_0} + 2^{k_0-1} + 2^{k_0-2} + ... = 2^{k_0+1}$$

r s'écrirait :

$$r = c_q 2^q + c_{q-1}2^{q-1} + ... + 2^{k_0+1} + 0 \times 2^{k_0} + 0 \times 2^{k_0-1} + ... = c_q 2^q + c_{q-1}2^{q-1} + ... + 2^{k_0+1}$$

c'est à dire que c_{k_0+1} passerait de 0 à 1 et tous les c_k pour $k \leq k_0$ passeraient de 1 à 0 et finalement le représentation binaire de r serait :

$$c_q c_{q-1} ... 100000 ...$$

Pour calculer c_k on divise d'abord r par 2^k, puis on prend la partie entière, et on prend le reste de la division par 2 de cette partie entière.
En effet :

$$r/2^k = ... + c_{k+1}2 + c_k 2^0 + c_{k-1}2^{-1} + ...$$

$c_{k-1}2^{-1} + ..$ est strictement inférieur à 1 puisque les c_k ne sont jamais tous égaux à 1 à partir d'un certain rang.
Pour $k \geq 0$ il suffit de calculer la représentation binaire de l'entier partie entière de r car on voit que la partie décimale n'intervient pas.

Exemple : 3.25

k				
...	...	terminé	...	...
2	3.25/4	0	0/2	0
1	3.25/2	1	1/2	1
0	3.25/1	3	3/2	1
-1	3.25× 2	6	6/2	0
-2	3.25× 4	13	13/2	1
-3	3.25× 8	26	26/2	0
...	...	terminé	...	...

terminé car on a que des nombres pairs
donc 3.25 s'écrit en binaire 11.01000... que des 0 exactement.

Autre exemple : .1

On part de $k = 0$ et on voit tout de suite qu'il n'y aura rien du côté des $k > 0$.

k	$.1/2^k$	$E(.1/2^k)$	reste $\left[E(.1/2^k)\right]/2$
0	.1/1=.1	0	0
-1	.1× 2=.2	0	0
-2	.2× 2=.4	0	0
-3	.4× 2=.8	0	0
-4	.8× 2=1.6	1	1
-5	1.6× 2=3.2	3	1

On voit alors que $3.2 = 3 + .2$: le 3 donnera toujours des multiples de 2 donc ne contribuera pas au reste, seul le .2 compte. Or le .2 a déjà été rencontré pour $k = -1$ donc à partir de $k = -6$ la séquence 0011 se répète à l'infini. La représentation binaire de .1 est donc 0.00011 0011 0011 ...

Exercice :

Montrer que la représentation binaire de .3 est également infinie et vaut 0.0 1001 1001 1001 ...

2.2 Ecriture des réels en binaire en mémoire

On utilisera les réels de type `double` qui, sur les PC utilisés en TD, sont écrits sur 8 octets (64 bits). Il existe aussi des réels écrits sur 4 octets (type `float` sur les PC utilisés en TD). Pour simplifier on va décrire le cas de 4 octets, mais tout sera directement transposable au cas de 8 octets.
On met 2^q en facteur dans l'expression du réel positif r quelconque vue ci-dessus :

$$r = \left(c_q 2^0 + c_{q-1}2^{-1} + ... + c_k 2^{k-q} + ...\right) \times 2^q$$

avec $k = q, q-1, ... -\infty$.
Puisque $c_q = 1$ et en écrivant en binaire le facteur entre parenthèses :

$$r = 1.c_{q-1}c_{q-2}...c_k... \times 2^q$$

Au total on a simplement décalé le point q fois vers la gauche dans l'expression binaire initiale de r et compensé en multipliant par 2^q.
La suite des c_k s'appelle la « mantisse binaire » .

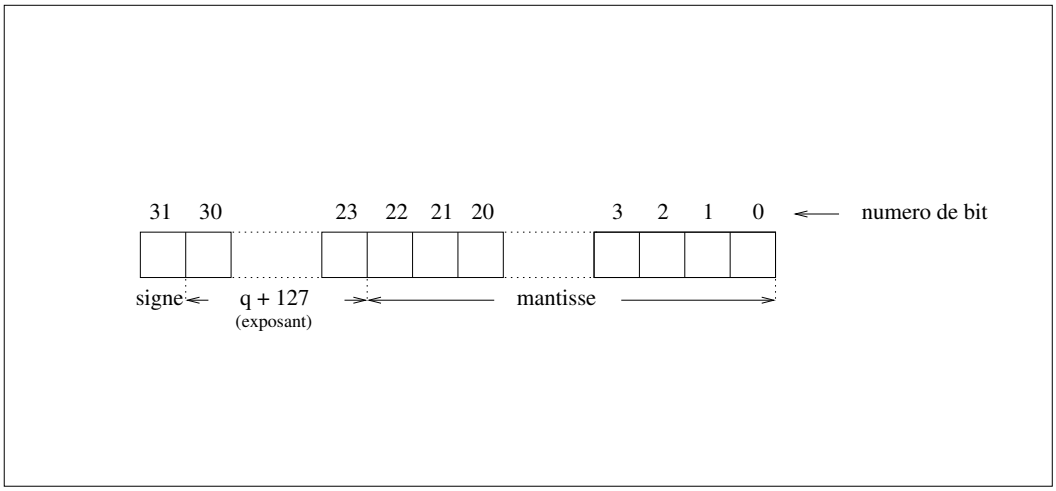


FIGURE 5 –

Le bit de gauche est pour le signe : 0 pour + et 1 pour −.
Les réels positifs et négatifs sont codés exactement de la même manière en dehors du bit de signe (contrairement au cas des entiers).
Dans les 8 suivants on met (en binaire) $q + 127$ (l'exposant est décalé de 127 pour n'avoir à stocker que des nombres positifs).
On pourrait donc à priori avoir $q+127$ variant de 0 à 255. Mais la valeur 0 est réservée pour signaler les nombres trop petits et la valeur 255 les nombres trop grands ou non définis. Ces deux valeurs 0 et 255 ont donc une signification spéciale. On a donc $1 \leq q + 127 \leq 254$, les valeurs extrêmes permises de q sont donc -126 et 127.
Dans les 23 restants on met les c_k.

2.2.1 Plus grand et plus petit nombre représentable

Cas général

Il découle de ce qui précède que la plus grande valeur absolue représentable est :
1.11...11 (23 fois 1 après le point)$\times\ 2^{127} = (2^{24}\text{-}1)\times\ 2^{104} \simeq 3.40282347\ 10^{38}$
et la plus petite :
1.00...00 (23 fois 0 après le point)$\times\ 2^{-126} = 2^{-126} \simeq 1.17549435\ 10^{-38}$.

Cas particulier des nombres sub-normaux

Tout ce paragraphe peut être sauté en première lecture.
En réalité du côté des petites valeurs il y a une astuce supplémentaire qui permet d'écrire des nombres jusqu'à une valeur absolue de $2^{-149} \simeq 1.4012984\ 10^{-45}$, mais avec de moins en moins de chiffres significatifs (nombres sub-normaux).
En effet lorsque l'exposant est nul mais la mantisse non nulle, on convient que le mot de 32 bits représente le nombre :

$$0.c_{q-1}c_{q-2}...c_k... \times 2^{-126} \quad \text{et non plus} \quad 1.c_{q-1}c_{q-2}...c_k... \times 2^q$$

La plus grande valeur absolue sub-normale est donc :
$0.11\ldots11$ (23 fois 1 après le point)$\times\ 2^{-126} = (2^{23}\text{-}1) \times\ 2^{-149} \simeq 1.17549421\ 10^{-38}$
et la plus petite :
$0.00\ldots01$ (22 fois 0 et 1 fois 1 après le point)$\times\ 2^{-126} = 2^{-149} \simeq 1.40129846\ 10^{-45}$

Récapitulation

Le tableau suivant résume les différents cas pour les nombres positifs, (le cas des nombres négatifs étant strictement identique au bit de signe près) :

	s i g n e	<q+127 > < mantisse >	valeur decimale
Plus grand normal	0	11111110 11111111111111111111	3.40282347 10+38
Plus petit normal	0	00000001 00000000000000000000	1.17549435 10-38
Plus grand sub-normal	0	00000000 11111111111111111111	1.17549421 10-38
Plus petit sub-normal	0	00000000 00000000000000000001	1.40129846 10-45
Zero positif	0	00000000 00000000000000000000	0.
Trop grand ou non def.	0	11111111	

2.2.2 Nombres représentables

L'ordinateur ne peut donc représenter de façon exacte qu'un nombre fini de nombres réels. Si on met de côté les nombres sub-normaux, il représente toutes les puissances de 2 de 2^{-126} à 2^{127} et, dans chaque intervalle $[2^p,2^{p+1}[$ [13], 2^{23} nombres puisque pour une valeur de l'exposant la mantisse peut prendre 2^{23} valeurs. Ces nombres sont régulièrement espacés de $(2^{p+1}-2^p)/2^{23} = 2^p/2^{23}$ puisque la mantisse augmente par pas de 2^{-23}. Dans l'intervalle immédiatement supérieur ils sont deux fois plus espacés (figure 3).
Dans l'intervalle $[2^{127},2^{128}[$ les nombres sont espacés de 2^{104}=20282409603651670423947251286016.

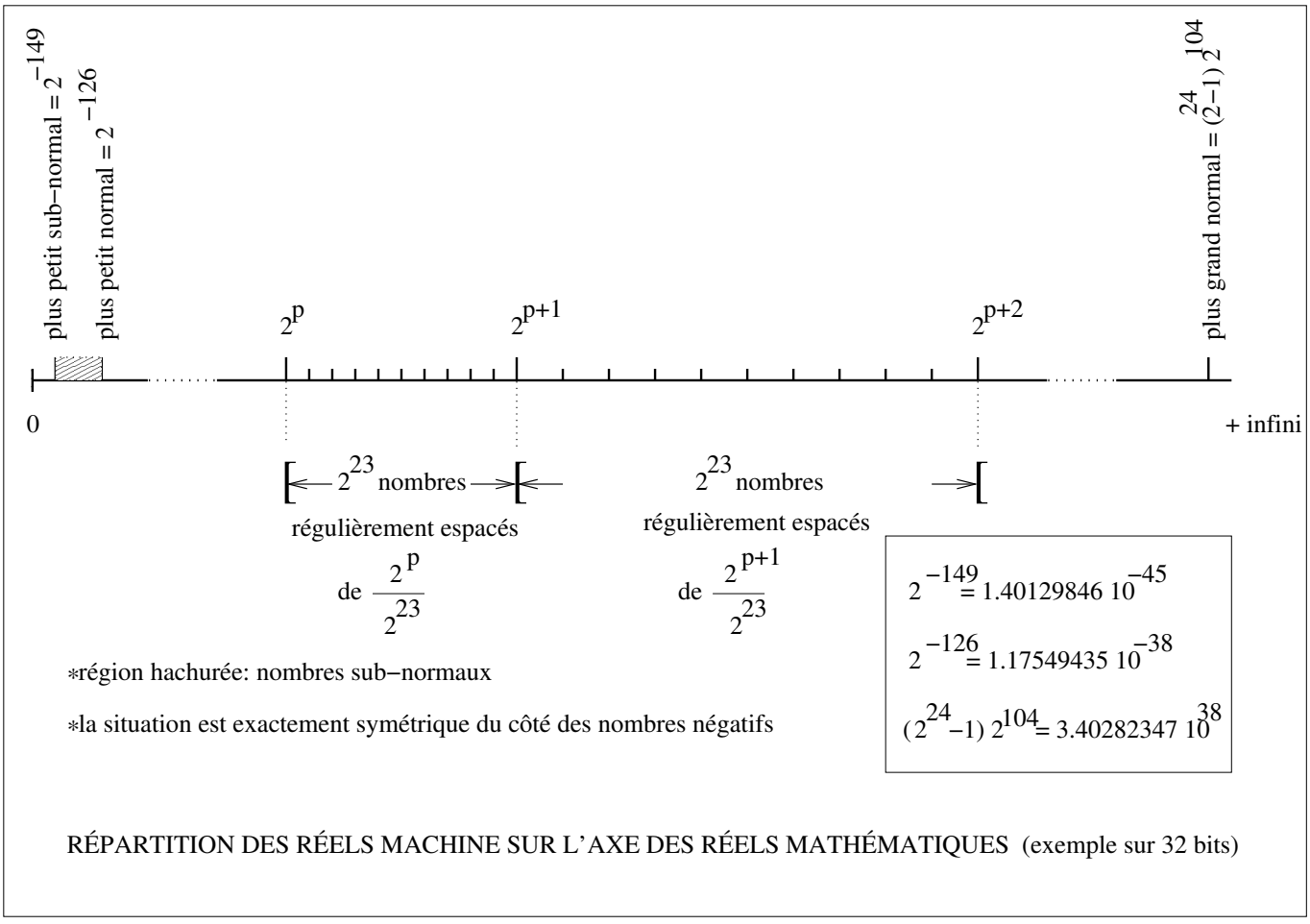


FIGURE 6 –

13. p pouvant donc varier de -126 à 127

Entre 2^{23} et 2^{24} les nombres sont espacés de 1 puis entre 2^{24} et 2^{25} espacés de 2. Ce qui fait que si on ajoute 1 à 2^{24} on obtient un nombre non représenté, puisque seuls le sont 2^{24} et $2^{24}+2$, et $2^{24}+1$ est tronqué à 2^{24}.
Ceci est illustré par le programme suivant [14] :

```
#include<iomanip>
#include<iostream>
using namespace std;
int main() {
  float x = 0., x1 = 1.;
  while(x<x1) {
    x = x1;
    x1 = x+1.;
  }
  cout << setprecision(16);
  cout << "On ne peut ajouter 1 en float que jusqu'à =" << x << endl;
  return 0;
}
```

dont le résultat est : *On ne peut ajouter 1 en float que jusqu'à =16777216*
C'est un peu comme si on ne pouvait soulever son pied que de trente centimètres et que l'on doive monter un escalier dont les marches deviennent de plus en plus hautes : il arriverait un moment où l'on resterait définitivement sur la même marche. On constate ainsi qu'avec un réel sur 4 octets on va moins loin pour énumérer les entiers de 1 en 1 qu'en utilisant un entier sur 4 octets (16777216 au lieu de 2147483647). C'est parce que de la place est perdue par l'exposant. Les réels sur 4 octets permettent de représenter des nombres bien plus grands que 2147483647 mais qui sont bien plus espacés que les entiers.

2.2.3 Troncature

Puisque les nombres représentables exactement sont en nombre fini un nombre quelconque est en général tronqué (à la valeur immédiatement inférieure, en valeur absolue), sauf s'il n'a que des zéros après le $23^{ième}$ chiffre de sa mantisse. Un réel ne peut être en général écrit qu'avec un nombre de chiffres significatifs exacts limité et il est important de connaître les limitations que cela entraîne.
On constate que des nombres qui s'écrivent très simplement en décimal comme .3 par exemple, n'ont pas une représentation exacte dans la machine : .3 est approché par .3000000119... . Donc dès l'introduction de ces nombres dans la machine il y a une certaine erreur, avant même tout calcul. Ensuite, après une opération le résultat sera lui-même tronqué ce qui entraîne une nouvelle erreur. Selon le calcul effectué les erreurs ainsi introduites peuvent se compenser plus ou moins, auquel cas l'erreur n'augmente que lentement au fil des opérations ou au contraire s'amplifier très rapidement et le résultat devient complètement faux.

Exercice :

Calculer les termes successifs de la suite :

$$u_0 = e - 1 \quad u_n = n u_{n-1} - 1 \quad e \text{ étant la base de la fonction exponentielle } 2.718..$$

Comparer les résultats obtenus avec des réels de 4 ou 8 octets.
Déterminer mathématiquement la limite de cette suite.

Remarque

Des nombres qui ont une représentation finie en binaire ont une représentation finie en décimale. Il n'y a donc pas de nouvelle erreur introduite quand on retraduit le résultat en décimal, à condition de prendre suffisamment de décimales.

Les nombres réels de type `double` sont écrits dans des mots de 8 octets (64 bits).

Le tableau suivant résume les limites et la précision des réels de type `float` et de type `double` avec les PC utilisés en TD :

14. Exemple emprunté à Francis Hecht

Type	Nb de bits signe	Nb de bits exposant	Nb de bits chiffres significatifs	Plus petite valeur absolue représentable	Plus grande valeur absolue représentable	Nb minimum de chiffres significatifs exacts en décimale
`float` (4 octets)	1	8	24	$\simeq 1.175\ 10^{-38}$ ($\simeq 1.401\ 10^{-45}$)	$\simeq 3.403\ 10^{38}$	6
`double` (8 octets)	1	11	53	$\simeq 2.225\ 10^{-308}$ ($\simeq 4.941\ 10^{-324}$)	$\simeq 1.798\ 10^{308}$	15

Dans le tableau ci-dessus les nombres sub-normaux sont placés entre parenthèses.

Rappel : $1.175\ 10^{-38}$ s'écrit en C : 1.175 e-38

Pour les nombres sub-normaux il n'y a pas de diagnostic, mais le nombre de chiffres significatifs exacts est moindre. Pour les valeurs absolues inférieures à la plus petite valeur sub-normale ou supérieures à la plus grande valeur normale il y a un diagnostic de dépassement.
Contrairement au cas des entiers il y a donc un diagnostic de dépassement pour les réels. Ces diagnostics seront vus en TD.

Remarques

Pour les nombres normaux le nombre de bits des chiffres significatifs est le nombre de bits de la mantisse plus un puisqu'ils s'écrivent 1.mantisse binaire $\times 2^q$.

Le mode de représentation des nombres que l'on vient de voir fait partie d'une norme (IEEE) qui se retrouve dans de nombreux langages de programmation.

Par contre le fait qu'en C un entier de type `int` ou un réel de type `float`, par exemple, soient écrits sur 4 octets ne fait pas partie de la norme du C. Cela peut varier d'un ordinateur à un autre et les valeurs données ici concernent les PC utilisés en TD. Mais le principe reste le même. On verra comment connaître précisément le nombre d'octets utilisés pour chaque type par un ordinateur donné.

3 Conclusion importante

Ce qui précède montre que si on interprète un mot de 4 octets par exemple, comme un entier ou comme un réel les deux résultats obtenus seront totalement différents.

Exercice

Se donner une suite quelconque de 4 octets et l'interpréter selon les règles vues précédemment soit comme un entier, soit comme un réel.

Annexe : suite $u_0 = e - 1 \quad u_n = nu_{n-1} - 1$

Programme :

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  float uf;
  double ud;
  int i, imax;
  imax = 100;
  cout << "n        float          double" << endl;
  for(uf = exp(1.)-1., ud = exp(1.)-1., i = 1; i <= imax; i++) {
    cout << i << "    " << uf << "    " << ud << endl;
    uf = i*uf-1.;
    ud = i*ud-1.;
  }
  return 0;
}
```

Résultat :

```
n        float          double
1    1.71828    1.71828
2    0.718282    0.718282
3    0.436564    0.436564
4    0.309691    0.309691
5    0.238765    0.238764
6    0.193824    0.193819
7    0.162943    0.162916
8    0.1406    0.140415
9    0.124802    0.123323
10    0.123215    0.109911
11    0.232147    0.0991122
12    1.55362    0.090234
13    17.6434    0.0828081
14    228.365    0.0765057
15    3196.1    0.0710802
16    47940.6    0.066203
17    767048    0.0592478
18    1.30398e+07    0.00721318
19    2.34717e+08    -0.870163
20    4.45962e+09    -17.5331
21    8.91923e+10    -351.662
22    1.87304e+12    -7385.9
23    4.12069e+13    -162491
24    9.47758e+14    -3.73729e+06
25    2.27462e+16    -8.96949e+07
26    5.68655e+17    -2.24237e+09
27    1.4785e+19    -5.83017e+10
28    3.99196e+20    -1.57415e+12
29    1.11775e+22    -4.40761e+13
30    3.24147e+23    -1.27821e+15
31    9.72441e+24    -3.83462e+16
32    3.01457e+26    -1.18873e+18
33    9.64661e+27    -3.80394e+19
34    3.18338e+29    -1.2553e+21
```

```
35    1.08235e+31    -4.26802e+22
36    3.78822e+32    -1.49381e+24
37    1.36376e+34    -5.37771e+25
38    5.04591e+35    -1.98975e+27
39    1.91745e+37    -7.56106e+28
40    inf    -2.94881e+30
41    inf    -1.17953e+32
42    inf    -4.83605e+33
43    inf    -2.03114e+35
44    inf    -8.73391e+36
45    inf    -3.84292e+38
46    inf    -1.72931e+40
47    inf    -7.95485e+41
48    inf    -3.73878e+43
49    inf    -1.79461e+45
50    inf    -8.79361e+46
51    inf    -4.3968e+48
52    inf    -2.24237e+50
53    inf    -1.16603e+52
54    inf    -6.17997e+53
55    inf    -3.33718e+55
56    inf    -1.83545e+57
57    inf    -1.02785e+59
58    inf    -5.85876e+60
59    inf    -3.39808e+62
60    inf    -2.00487e+64
61    inf    -1.20292e+66
62    inf    -7.33782e+67
63    inf    -4.54945e+69
64    inf    -2.86615e+71
65    inf    -1.83434e+73
66    inf    -1.19232e+75
67    inf    -7.86931e+76
68    inf    -5.27243e+78
69    inf    -3.58526e+80
70    inf    -2.47383e+82
71    inf    -1.73168e+84
72    inf    -1.22949e+86
73    inf    -8.85234e+87
74    inf    -6.46221e+89
75    inf    -4.78203e+91
76    inf    -3.58653e+93
77    inf    -2.72576e+95
78    inf    -2.09883e+97
79    inf    -1.63709e+99
80    inf    -1.2933e+101
81    inf    -1.03464e+103
82    inf    -8.3806e+104
83    inf    -6.87209e+106
84    inf    -5.70383e+108
85    inf    -4.79122e+110
86    inf    -4.07254e+112
87    inf    -3.50238e+114
88    inf    -3.04707e+116
89    inf    -2.68142e+118
90    inf    -2.38647e+120
91    inf    -2.14782e+122
```

```
92    inf    -1.95452e+124
93    inf    -1.79816e+126
94    inf    -1.67228e+128
95    inf    -1.57195e+130
96    inf    -1.49335e+132
97    inf    -1.43362e+134
98    inf    -1.39061e+136
99    inf    -1.3628e+138
100    inf    -1.34917e+140
```

4. Variables, constantes, types, opérateurs

1 Variables, constantes, types

1.1 Variables et constantes

Les variables et les constantes peuvent être considérées comme des registres de mémoire pour lesquels l'utilisateur choisit un nom. Ce nom doit être constitué de caractères dits « alphanumériques » c'est à dire les vingt-six lettres de l'alphabet, les dix chiffres et le caractère souligné. Les majuscules sont autorisées et sont distinguées des minuscules : `x` et `X` ne représentent pas la même variable. Un nom ne peut pas commencer par un chiffre et peut comporter au plus trente et un caractères.
Chaque variable ou constante a un type, défini par une déclaration. Dans ce cours on se limite aux trois types principaux :

- entier, déclaré par `int x`
- réel, déclaré par `double x`
- caractère, déclaré par `char x`

Sur les PC utilisés en TD le type `int` est écrit sur 4 octets, le type `double` sur 8, le type `char` sur 1.
Pour indiquer qu'un nom désigne une constante et non pas une variable, on ajoute le mot clé `const` dans la déclaration de `x` et on lui attribue sa valeur [15] : `const int x = 2`

Remarque

Il faut savoir qu'il existe aussi des entiers sur 1 octet, sur 2 octets, avec ou sans signe, des réels sur 4 octets (type `float`), même si nous ne nous servirons pas de ces types.

1.2 Constantes explicites

Ce sont des nombres ou des caractères.

un nombre entier se note par exemple : `123`
un nombre réel de type `double` : `1.23` ou `0.123e1` ou `12.3e-1` etc. pour le même nombre réel 1.23
un caractère : `'a'` (entre « simples quotes ») [16]

Pour les réels :

`1.6e-19` est donc la notation informatique de $1.6 \cdot 10^{-19}$
on met un point même s'il s'agit d'un nombre n'ayant que des chiffres décimaux nuls, pour bien distinguer entre entiers et réels. On écrit le réel 123 : `123.`

2 Attribution de valeur : le signe =

L'instruction `y = x` a un sens différent de celui qu'a en mathématiques la proposition $y = x$. Elle signifie qu'on met dans le registre de mémoire situé à gauche du signe = (désigné ici par `y`) le contenu qui se trouve dans le registre de mémoire situé à droite du signe = (ici désigné par `x`) à l'instant où l'instruction est exécutée. Par exemple :

`x = 1` `x` vaut 1
`y = x` `x` vaut 1 `y` vaut 1
`x = 2` `x` vaut 2 `y` vaut 1

On peut noter aussi sur l'exemple suivant qu'en C le signe = n'a pas non plus exactement la même signification que dans un langage comme Maple :

15. Le compilateur C ignore des espaces supplémentaires : pour lui `x=2` et `x = 2` sont équivalents. Dans ce poly on préfère mettre des espaces autour des signes = etc. pour améliorer la lisibilité des programmes.
16. On verra dans la suite que les « chaînes de caractères » se notent entre « doubles quotes » : `"abc"`

	C	Maple

`x=1`
`y=x`
`x=2`
	`x` vaut 1
	`x` et `y` valent 1
	`x` vaut 2, `y` vaut 1
		idem
		idem
		idem

`y=x`
`x=1`
`x=2`
	`x` et `y` ont une même valeur bien déterminée, mais inconnue
	`x` vaut 1, `y` garde sa valeur inconnue
	`x` vaut 2, `y` garde sa valeur inconnue
		`x` et `y` n'ont pas de valeur bien déterminée et sont égaux
		`x` vaut `1`, `y` vaut `1`
		`x` vaut `2`, `y` vaut `2`

Dans le second tableau on suppose qu'aucune valeur n'a été attribuée à `x` précédemment.
En C comme en Maple la règle est que `y` vaut ce que vaut `x` au moment où on écrit `y=x`. La différence est qu'en Maple une variable peut très bien ne pas avoir de valeur bien définie (comme en mathématiques) alors qu'en C une variable vaut toujours le contenu de la mémoire qu'elle désigne et donc a nécessairement une valeur bien particulière même si celle-ci n'a aucun sens.

Remarque

D'après ce qui précède une constante ne pourra pas figurer à gauche d'un signe = hormis dans sa déclaration.

3 Opérateurs

De façon générale, un opérateur est une règle qui fait correspondre à n éléments $x_1, x_2, ..., x_n$ donnés un élément y :

$$x_1, x_2, ..., x_n \longrightarrow y$$

Si $n = 1$ l'opérateur est dit « unaire »
Si $n = 2$ il est dit « binaire »

Exemple d'opérateur unaire :

$$x \longrightarrow y = -x$$

Exemples d'opérateurs binaires :

$$\begin{array}{lcll} x_1, x_2 & \longrightarrow & y = x_1 x_2 & \text{multiplication} \\ x_1, x_2 & \longrightarrow & y = 0 \text{ si } x_1 \neq x_2 & \text{comparaison} \\ & & y = 1 \text{ si } x_1 = x_2 & \end{array}$$

3.1 Opérateurs arithmétiques, règles de priorité

Les opérateurs arithmétiques sont présentés dans le tableau suivant :

	Signification mathématique	
Opérateur	Entiers	Réels
+	addition	addition
-	soustraction	soustraction
*	multiplication	multiplication
/	partie entière de la division	division
%	reste de la division	non défini

Exemple

2/3 vaut 0
2./3. vaut 0.333333
2%3 vaut 2

Dans une suite d'opérations (formule) il y a des priorités :
- les calculs effectués en premier sont ceux placés entre parenthèses, en commençant par les parenthèses les plus internes.
- ensuite sont effectués : * ou / ou % puis + ou -

* / et % ont la même priorité
+ et - ont la même priorité

A priorité égale les opérations sont effectuées de la gauche vers la droite.

Exemple

$$\frac{a+b}{c+d} \qquad \text{sera écrit :} \qquad \texttt{(a+b)/(c+d)}$$

alors que :

`a+b/c+d` donne : $a+\frac{b}{c}+d$

`(a+b)/c+d` donne : $\frac{a+b}{c}+d$

et :

`a+b/(c+d)` donne : $a+\frac{b}{c+d}$

Par ailleurs :

$$\frac{a}{bc} \qquad \text{peut s'écrire indifféremment :} \qquad \texttt{a/(b*c)} \text{ ou } \texttt{a/b/c}$$

Exercice : exemple d'utilisation de l'opérateur %

On convient que :

Lundi=1, Mardi=2, ..., Dimanche=7

On veut déterminer quel jour j de la semaine est le $n^{\text{ème}}$ jour de l'année, sachant que le premier Janvier est le jour p de la semaine. On fournit donc une valeur de n comprise entre 1 et 365 (ou 366) et une valeur de p comprise entre 1 et 7 et on cherche la valeur de j, comprise entre 1 et 7. Ecrire une expression composée de `n` et `p` et dont le résultat est `j`.

Réponse : `j=(n+p-2)%7+1`

Si l'on veut utiliser des opérateurs un peu moins élémentaires il est nécessaire d'utiliser la bibliothèque mathématique du C en incluant la directive :

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

Pour les entiers on dispose alors de la valeur absolue de n avec `abs(n)`.
Pour les réels les fonctions les plus courantes sont présentées dans le tableau ci-dessous :

Notation mathématique	x^y	$\sqrt{x}$	$\exp x$	$\ln x$	$\log x$	$\lvert x\rvert$	$\sin x$	$\cos x$	$\tan x$
Notation C	`pow(x,y)`	`sqrt(x)`	`exp(x)`	`log(x)`	`log10(x)`	`fabs(x)`	`sin(x)`	`cos(x)`	`tan(x)`

Toutes ces fonctions ont des valeurs et des arguments de type `double`.

Remarques

Les opérations non définies telles que division par 0, racine carrée d'un nombre négatif, élévation d'un nombre négatif à une puissance réelle, etc., donnent lieu à un diagnostic en cours d'exécution.

Si une opération aboutit à un dépassement de capacité :
— pour les entiers il n'y a pas de diagnostic et le résultat est complètement aberrant
— pour les réels il y a un diagnostic mais le calcul se poursuit et le résultat peut être complètement aberrant.

L'instruction `a += b` est équivalente à `a = a+b`, l'instruction `i++` est équivalente à `i = i+1`.
Il existe de même `a -= b`, `a *= b`, `a /= b`, `a %= b` [17].

3.2 Mélange des types entier et réel

3.2.1 Conversion de type dans une affectation

Lors d'une affectation :

```
y = x;
```

la valeur de `x` est convertie dans le type de `y` avant d'être mise dans `y`. Supposons que `x` soit un `double` : si `y` est un `double` le résultat final sera bien un `double` mais si `y` est un `int` le résultat final sera un `int` et il y aura donc troncature à l'entier de valeur absolue inférieure ou égale.

3.2.2 Mélange de types dans une expression

On peut dans une expression mélanger les types entier et réel. Mais il faut l'éviter le plus possible.
Quand le calcul peut être effectué uniquement avec des entiers, il ne faut pas introduire inutilement de réels : les opérations sont plus rapides avec les entiers et ne sont pas affectées d'erreur de troncature (sauf cas de la division).
Quand le calcul comprend nécessairement à la fois des entiers et des réels, il faut tenir compte de la règle suivante :

lors d'une opération (+ - * /) entre un entier et un réel, l'entier est converti automatiquement en réel avant l'opération et le résultat est toujours réel (on dit que le réel l'emporte toujours).

Il faut aussi tenir compte de la conversion lors de l'affectation en examinant à quel type de variable ce résultat est affecté.

Exemple

17. Attention, il peut y avoir des cas à traiter avec précaution : par exemple `a`, `b` et `c` étant des entiers, `a *= b/c` est équivalent à `a = a*(b/c)`, ce qui donne un résultat différent de `a = a*b/c` si `b` n'est pas divisible par `c`.

```
#include<iostream>
using namespace std;
int main() {
  int i, j = 2, k = 3;
  double x, y = 2., z = 3.;
  i = j/k; cout << "Resultat : " << i << endl;
  x = y/z; cout << "Resultat : " << x << endl;
  x = j/z; cout << "Resultat : " << x << endl;
  x = y/k; cout << "Resultat : " << x << endl;
  i = j/z; cout << "Resultat : " << i << endl;
  i = y/k; cout << "Resultat : " << i << endl;
  x = j/k; cout << "Resultat : " << x << endl;
  i = y/z; cout << "Resultat : " << i << endl;
  return 0;
}
```

donne :

Resultat : 0
Resultat : 0.666667
Resultat : 0.666667
Resultat : 0.666667
Resultat : 0
Resultat : 0
Resultat : 0
Resultat : 0

Les fonctions mathématiques indiquées ci-dessus [18], réalisent la conversion automatique du type des arguments vers le type `double`. Ainsi :

```
...
int i; double y;
...
... = sqrt(i);
... = sqrt(y);
...
```

`i` est converti en `double` avant le calcul de la racine carrée, `y` est pris tel quel, et le résultat est `double`.

Remarque (à sauter en première lecture)

En mathématiques, x^y n'est défini pour x négatif que si y est un nombre rationnel de la forme p/q avec p et q entiers premiers entre eux et q impair. Si x est négatif x^y est donc défini pour y entier (positif ou négatif) et pour $y = 2/3$ par exemple, mais pas pour $y = 3/2$. En C les arguments de `pow(x,y)` sont toujours réels [19] au sens informatique et on peut se demander ce qu'il advient lorsque x est négatif. Si y est entier au sens mathématique, c'est à dire vaut 1.000..., 2.000..., (ou -1.000..., -2.000...) etc. la fonction `pow` est capable de calculer `pow(x,y)`, mais dans les autres cas non. Elle ne peut reconnaître que y est un rationnel du type précédent, ce qui n'est pas étonnant. Elle refuse de calculer $(-1.2)^{3/2}$, ce qui est normal puisque cette quantité n'est pas définie, mais elle refuse aussi de calculer $(-1.2)^{2/3}$ qui est pourtant bien définie. Ceci est illustré par le programme suivant :

```
#include<iostream>
#include <math.h>
using namespace std;
int main() {
  int i, j;
  double x, y, z;
  /* pow(-4.5,2./3.) ne peut etre calcule : */
  x = 4.5; y = 2./3.; z = 3./2.;
```

18. comme les autres fonctions qui seront étudiées ultérieurement
19. éventuellement après une conversion

```
  cout << endl;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y)
       << "   pow(" << x << "," << z << ")=" << pow(x,z) << endl;
  x = -4.5;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y)
       << "   pow(" << x << "," << z << ")=" << pow(x,z) << endl << endl;
  /* mais pow(-4.5,3.) est correctement calcule : */
  x = 4.5; y = 3.;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y) << endl;
  x = -4.5;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y) << endl << endl;
  /* et la conversion entier-reel se fait bien : */
  i = 7; j = 3;
  cout << "pow(" << i << "," << j << ")=" << pow(i,j) << endl;
  i = -7;
  cout << "pow(" << i << "," << j << ")=" << pow(i,j) << endl << endl;
  return 0;
}
```

qui donne le résultat :

pow(4.5,0.666667)=2.72568 pow(4.5,1.5)=9.54594
pow(-4.5,0.666667)=nan pow(-4.5,1.5)=nan
pow(4.5,3)=91.125
pow(-4.5,3)=-91.125
pow(7,3)=343
pow(-7,3)=-343

3.2.3 Opérateur de conversion de type (...)

Soit deux variables :

```
int i; double x;
```

l'expression `(double)i` représente la valeur de `i` transformée en `double`
l'expression `(int)x` représente la valeur de `x` transformée en `int`

Ces expressions ne sont pas des variables, elles ne peuvent pas figurer à gauche d'un signe `=`.
De façon plus générale l'expression `(t)y` représente la valeur de `y` transformée dans le type `t`.
`(t)` s'appelle l'opérateur de conversion de type vers le type `t` (opérateur « cast » en anglais).
Ainsi :

```
...
int i, j;
i = 2; j = 3;
cout << i/j << " " << (double)i/j << endl;
...
```

donne : *0 0.666667*

3.3 Exemple

Formule de Planck :

$$u(\nu) = \frac{8\pi h\nu^3}{c^3} \frac{1}{e^{\frac{h\nu}{kT}} - 1}$$

peut par exemple s'écrire :

```
#include<iostream>
#include<iomanip>
#include<math.h>
using namespace std;
int main() {
  const double h = 6.62618e-34, k = 1.38066e-23, c = 2.99792e8;
  double b, nu, t, u;
  nu = 1.e15; t = 5.e3;
  b = h*nu/k/t;
  u = 8.*M_PI*h*pow(nu/c,3)/(exp(b)-1.);
  cout << "Resultat : " << u << endl;
  return 0;
}
```

`M_PI` est une constante définie dans `math.h` qui donne la valeur de π.
Plus il y a de niveaux de parenthèses, plus, en général, la formule est difficile à lire. Il faut donc éviter de mettre des parenthèses inutiles.
Cependant dans quelques cas une parenthèse inutile rend au contraire la formule plus facile à lire.
Il faut aussi utiliser des variables intermédiaires, surtout si des sous-formules identiques se répètent dans la formule. Il ne faut pas non plus pulvériser la formule en multipliant les variables intermédiaires, car la formule s'étend alors en hauteur au lieu de s'étendre en largeur. Il n'y a pas de règle stricte, c'est une question d'appréciation personnelle du programmeur, l'objectif étant toujours la lisibilité, pour gagner du temps de lecture et diminuer le risque d'erreur.

3.4 Opérateurs de comparaison

Ce sont `<` `<=` `>` `>=` `==` `!=`, la signification des quatre premiers est explicite, les deux derniers désignant respectivement « égal à » et « différent de » . L'expression `x > y` a la valeur entière 1 si elle est vraie et 0 si elle est fausse, et de même pour les autres opérateurs de comparaison.

Exemple

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y;
  x = 1.; y = 2.;
  i = x>y; cout << "i=" << i << endl;
  i = x<y; cout << "i=" << i << endl;
  return 0;
}
```

Affiche à l'écran le résultat :

i=0
i=1

Remarque

Pour les caractères la comparaison teste l'ordre alphabétique (ne fonctionne pas pour les chaînes de caractères).

Remarque

Les tests entre réels souffrent de l'imprécision attachée à la représentation des réels. Par exemple, sur les PC utilisés en TD, le programme suivant :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y;
  x = 0.5; y = 0.25;
  i = (x*x == y); cout << i << endl;
```

```
  x = 0.3; y = 0.09;
  i = (x*x == y); cout << i << endl;
  return 0;
}
```

donne comme résultat :

1
0

Il ne faut pas oublier cela quand on fait des comparaisons entre réels.

3.5 Opérateurs logiques

Ce sont `&&`, `||` et `!`, correspondant respectivement à « et » , « ou » et « non » .
Exemple

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y, z;
  x = 1.; y = 2.; z = 3.;
  i = x<y && y<z; cout << "i=" << i << endl;
  i = x>y || y==z; cout << "i=" << i << endl;
  i = !(x>y || y>z); cout << "i=" << i << endl;
  return 0;
}
```

Donne :

i=1
i=0
i=1

3.6 Priorités des opérateurs arithmétiques, de comparaison et logiques

Dans une expression qui comprend simultanément des opérateurs arithmétiques, de comparaison et logiques, l'ordre de priorité décroissante est : arithmétique, comparaison, logique.
Entre les opérateurs de comparaison `<`, `<=`, `>`, `>=` ont tous la même priorité, et `==` et `!=` la priorité immédiatement inférieure.
`&&` est prioritaire sur `||`.
Les termes entre parenthèses sont évalués en premier en commençant par les parenthèses les plus internes et à priorité égale les opérations sont effectuées dans l'ordre de l'écriture.

Exemple

`n1, n2, n3` étant par exemple des entiers :

`n1+n2 > n3` l'addition est effectuée avant la comparaison

`n1+n2 > n3 || n1-n2 < n3` addition et soustraction sont effectuées d'abord puis les deux comparaisons, puis le ou.
Il peut être utile de mettre des parenthèses même si elles ne sont pas nécessaires, pour rendre l'expression plus lisible :
`((n1+n2) > n3) || ((n1-n2) < n3)`

Liste des opérateurs mentionnés jusqu'ici, par ordre de priorité décroissante d'une ligne à l'autre :

Opérateurs
`()`
`!`
`* / %`
`+ -`

```
<   <=   >   >=
   ==   !=
     &&
     ||
```

Exercice : réduction au musée

Un individu peut obtenir une réduction s'il remplit une des conditions suivantes :
— il a moins de 18 ans
— il a moins de 25 ans et fait des études
— il a moins de 25 ans et ses deux parents ne travaillent pas
On définit les variables de type `int` suivantes :

`age` qui vaut l'âge de l'individu
`etu` qui vaut 1 s'il fait des études, 0 sinon
`pt` qui vaut 1 si le père travaille, 0 sinon
`mt` qui vaut 1 si la mère travaille, 0 sinon

Ecrire une expression composée de ces quatre variables qui vaut 1 s'il a droit à la réduction, 0 sinon.

Réponse : `(((mt¡ && pt¡) || etu) && age<25) || age<18`

Quelles parenthèses peut-on supprimer dans la réponse précédente ?

Réponse : les plus externes et les plus internes

5. Tests et boucles

Dans les programmes vus jusqu'ici, les instructions sont toutes exécutées une fois et une seule et ce, dans l'ordre où elles sont écrites dans le fichier. Les tests permettent de n'exécuter qu'une partie des instructions, en fonction des valeurs que prennent certaines variables, et les boucles de répéter certaines instructions.

1 Tests

1.1 Alternative `if`

On rappelle que des instructions forment un bloc si elles sont comprises entre deux accolades.
L'instruction `if` permet de faire exécuter un bloc d'instructions si et seulement si une certaine condition est réalisée et un autre bloc si et seulement si elle ne l'est pas.

1.1.1 Exemple

On veut calculer la fonction $\sin x/x$. Il faut faire un cas particulier pour $x = 0$. On écrit donc :

```
...
double x, f;
...
x = ...;
if(x == 0.) {
  f = 1.;
}
else {
  f = sin(x)/x;
}
...
```

Dans cet exemple particulier les blocs d'instructions à effectuer dans l'un ou l'autre cas se réduisent chacun à une seule instruction.

1.1.2 Forme générale

```
if(expression) {
  instructions;
}
else {
  instructions;
}
```

Si `expression` est vraie (donc vaut 1), le premier bloc d'instructions est exécuté puis le second est sauté et le cours normal du programme reprend à la première instruction qui suit le second bloc.
Si `expression` est fausse (donc vaut 0), le premier bloc est sauté, le second est exécuté puis, le cours normal du programme continue à la première instruction qui suit le second bloc.

Remarques

1) Si un des blocs ne contient qu'une instruction on n'est pas obligé de mettre les `{ }`
2) S'il n'y a pas d'instructions à exécuter lorsque `expression` est fausse on peut ne pas mettre le `else`
3) Il est possible d'inclure un second `if` dans un des blocs d'instructions, d'en inclure un troisième dans le second et ainsi de suite sans limitation. Il suffit de respecter la règle d'inclusion : un `else` se rapporte toujours au dernier `if` rencontré auquel un `else` n'a pas encore été attribué.

Exemples

1) Conformément à la remarque 1) le `if` du programme qui calcule $\sin x/x$ peut être écrit sans accolades :

```
if(x == 0.)
  f = 1.;
else
  f = sin(x)/x;
```

2) Pour illustrer la remarque 2) on calcule la fonction d'Heaviside :

$$\begin{aligned} H(x) &= 0 \quad \text{pour } x < 0 \\ H(x) &= 1 \quad \text{pour } x \geq 0 \end{aligned}$$

On peut l'écrire de la façon suivante :

```
...
if(x < 0.)
  H = 0.;
else
  H = 1.;
...
```

Mais aussi, ce qui peut être plus simple dans certains cas :

```
...
H = 0.
...
if(x >= 0.)
  H = 1.;
...
```

3) Pour illustrer la remarque 3) on calcule le point d'intersection de deux droites dans le plan, d'équations :

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

Il faut, avant d'écrire le programme, examiner les différents cas possibles :

$a = 0$ et $b = 0$ ou $d = 0$ et $e = 0$	une des deux droites n'est pas definie	
$ae - db = 0$	$af - dc = 0$	les droites sont confondues
	$af - dc \neq 0$	les droites ne se coupent pas
$ae - db \neq 0$	les deux droites se coupent	

Ce qui peut s'écrire :

```
#include<iostream>
using namespace std;
int main() {
  double a, b, c, d, e, f, det, det1, det2;
  a = ...; b = ...; c = ...; d = ...; e = ...; f = ...;
  if(a == 0. && b == 0. || d == 0. && e == 0.)
    cout << "Au moins une des deux droites n'est pas definie" << endl;
  else {
    det = a*e-d*b;
    det1 = a*f-d*c;
    if(det == 0.) {
      if(det1 == 0.)
        cout << "Les deux droites sont confondues" << endl;
      else
        cout << "Les deux droites ne se coupent pas" << endl;
    }
    else {
```

```
        det2 = b*f-e*c;
        cout << "Solution unique : x=" << det2/det << "  y=" << det1/det << endl;
      }
    }
    return 0;
  }
```

On remarque que, lorsque comme ici, la structure devient un peu compliquée, il faut indenter [20] avec soin pour que le programme soit lisible.

1.1.3 Choix multiple

On considère par exemple trois intervalles disjoints sur l'axe des réels : $[x_1^{\min}, x_1^{\max}]$, $[x_2^{\min}, x_2^{\max}]$, $[x_3^{\min}, x_3^{\max}]$. Etant donné une variable réelle `x`, on veut qu'une variable entière `i` vaille :

1 si `x` appartient à l'intervalle $[x_1^{\min}, x_1^{\max}]$
2 si `x` appartient à l'intervalle $[x_2^{\min}, x_2^{\max}]$
3 si `x` appartient à l'intervalle $[x_3^{\min}, x_3^{\max}]$
0 sinon.

Si on applique strictement ce qui précède on doit écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, x1min = -1., x2min = 0.5, x3min = 2.;
  double x1max = -0.5, x2max = 0.8, x3max = 2.9;
  x = 0.7;
  if(x >= x1min && x <= x1max)
    i = 1;
  else {
    if(x >= x2min && x <= x2max)
      i=2;
    else {
      if(x>=x3min && x<=x3max)
        i=3;
      else
        i=0;
    }
  }
  cout << "i=" << i << endl;
  return 0;
}
```

Mais en réalité les accolades pour imbriquer les `if ... else` les uns dans les autres ne sont pas nécessaires et on peut écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, x1min = -1., x2min = 0.5, x3min = 2.;
  double x1max = -0.5, x2max = 0.8, x3max = 2.9;
  x = 0.7;
  if(x >= x1min && x <= x1max)
    i=1;
  else if(x >= x2min && x <= x2max)
```

20. utiliser l'indentation automatique de *emacs*

```
    i=2;
  else if(x >= x3min && x <= x3max)
    i=3;
  else
    i=0;
  cout << "i=" << i << endl;
  return 0;
}
```

Remarque

S'il y a plusieurs instructions à exécuter dans le cas où un `if` ou un `else` est réalisé[21], alors il est évidemment nécessaire de mettre ces instructions entre accolades, aussi bien dans le premier cas que dans le second.

1.2 Aiguillage `switch`

Dans le cas du choix multiple le `switch` peut être plus pratique que le `if` mais n'est pas toujours utilisable. Supposons qu'on doive faire exécuter le groupe d'instructions (1) si la variable entière `n` vaut 1, le groupe d'instructions (2) si elle vaut 2, le groupe d'instructions (3) si elle vaut 3 et le groupe d'instructions (4) sinon. Avec un `if` il faut écrire :

```
...
if(n == 1) {
  instructions_1;
}
else if(n == 2) {
  instructions_2;
}
else if(n == 3) {
  instructions_3;
}
else {
  instructions_4;
}
...
```

Avec un `switch` cela s'écrit :

```
...
switch(n) {
case 1 :
  instructions_1;
  break;
case 2 :
  instructions_2;
  break;
case 3 :
  instructions_3;
  break;
default :
  instructions_4;
}
...
```

La forme générale du `switch` est la suivante :

21. au lieu d'une seule comme ici : `i = ...`

```
switch(expression) {
case constante_1 :
  instructions_1;
  break;
case constante_2 :
  instructions_2;
  break;
...
default :
  instructions;
}
```

Deux cas se présentent :

- `expression` vaut une des `constante_i` : alors les `instructions_i` sont exécutées. Le `break` n'est pas obligatoire : si `instructions_i` se termine par un `break` on passe ensuite après l'accolade de fin du bloc et le `switch` est terminé, sinon ce sont les `instructions_(i+1)` qui sont exécutées et ainsi de suite jusqu'à rencontrer un `break`
- `expression` ne vaut aucune des `constante_i` : ce sont les `instructions` situées après `default` qui sont exécutées, puis on sort du `switch`.

Remarques

`expression` doit être de type `int` ou `char`. C'est pourquoi l'exemple précédent des trois intervalles ne peut pas être traité par un `switch`, sinon d'une façon qui n'apporte pas de simplification par rapport à un `if`.

Les `constante_i` peuvent être explicites ou non.

Certaines `instructions_i` peuvent être vides (y compris de `break`). Ceci permet d'avoir plusieurs `constante_i` conduisant aux mêmes `instructions_j`.

Exemple 1 :

```
...
char rep;
...
cout << "Doit-on calculer le perimetre (p), la surface (s) ou le volume (v) ?" << endl;
cin >> rep; // met dans la variable caractère rep la réponse frappée au clavier
switch(rep) {
case 'p':
  ...
  break;
case 's':
  ...
  break;
case 'v':
  ...
  break;
default :
  ...
}
```

Exemple 2 :

Soit deux disques, l'un centré au point de coordonnées a_1, b_1 et de rayon r_1, l'autre centré en a_2, b_2, de rayon r_2. On veut déterminer si un point de coordonnées x, y appartient aux deux disques, à l'un des deux seulement ou à aucun des deux.

```
/* Position d'un point par rapport a deux disques */
#include<iostream>
using namespace std;
int main() {
```

```
  double a1 = -3., b1 = 5., r1 = 2., a2 = 1.5, b2 = 4., r2 = 5.;
  double x = 0., y = 10., d1, d2;
  d1 = (x-a1)*(x-a1) + (y-b1)*(y-b1);
  d2 = (x-a2)*(x-a2) + (y-b2)*(y-b2);
  switch((d1 <= r1*r1) + (d2 <= r2*r2)) {
  case 0 :
    cout << "Le point est exterieur aux deux disques" << endl;
    break;
  case 1 :
    cout << "Le point est interieur a l'un des disques et exterieur a l'autre" << endl;
    break;
  case 2 :
    cout << "Le point est interieur aux deux disques" << endl;
    break;
  }
  return 0;
}
```

Dans ce cas il est inutile de mettre un cas `default` puisque la valeur de l'expression ne peut jamais être différente de 0, 1 ou 2.

2 Boucles

2.1 Boucle `while`

2.1.1 Exemple

Supposons qu'on veuille calculer les puissances successives de 2. Dès que l'on veut en calculer plus de trois il serait lourd d'écrire :

```
...
x = 1;   cout << x << endl;
x = x*2; cout << x << endl;
x = x*2; cout << x << endl;
...
```

On utilise alors une boucle. Par exemple le `while` :

```
...
x = 1;
while(x <= 1000) {
  x = x*2; cout << x << endl;
}
...
```

calcule les puissances de 2 inférieures ou égales à 2000. Tant que `x` est inférieur ou égal à 1000 on effectue les instructions du bloc. Sinon on passe après le bloc.

2.1.2 Forme générale

```
while(expression) {
  instructions;
}
```

La première fois que le `while` est rencontré `expression` est évaluée :

— si elle est fausse on passe directement après le bloc sans être entré une seule fois dans la boucle

— si elle est vraie les instructions du bloc sont exécutées puis on revient au `while` où l'expression est de nouveau évaluée. Tant qu'elle reste vraie on ré-exécute le bloc jusqu'à ce qu'`expression` devienne fausse. Si elle ne devient jamais fausse, la boucle est exécutée indéfiniment et le programme ne s'arrête jamais[22], sauf si on utilise une sortie de boucle à l'aide de l'instruction `break`, présentée à la section suivante.

2.1.3 `break` et `continue`

Les instructions `break` et `continue` se placent nécessairement dans une boucle :

`break` fait sortir définitivement de la boucle et passer à l'instruction qui la suit

`continue` interrompt le tour en cours et fait passer au tour suivant

En cas de boucles imbriquées `break` et `continue` concernent la boucle la plus interne les contenant.

Remarque

`break` et `continue` sont utilisables non seulement dans la boucle `while` mais aussi dans toutes celles vues dans la suite.

2.1.4 Exemple de boucle infinie

La boucle :

```
while(1) {
...
}
```

ne s'arrête, à priori, jamais, puisque l'expression testée par le `while` vaut toujours 1 et est donc toujours vraie. En ajoutant un `break` elle peut être utilisée comme dans l'exemple suivant :

```
/* Calcul du carre d'un nombre */
#include<iostream>
using namespace std;
int main() {
  int n;
  cout << "Répondre 0 pour terminer l'exécution" << endl;
  while(1) {
    cout << "n=? "; cin >> n;
    if(n == 0)
      break;
    cout << n << " au carré = " << n*n << endl;
  }
  return 0;
}
```

2.1.5 Exemple de boucle utilisant un `continue`

On veut faire la liste des nombres inférieurs ou égaux à 1000 divisibles par 2 ou 3 :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  i = 1;
  while(i <= 1000) {
    if(i%2 == 0) {
      cout << i << endl;
      i++;
      continue;
    }
```

22. arrêter alors l'exécution par *Ctrl-c*

```
    if(i%3 == 0)
      cout << i << endl;
    i++;
  }
  return 0;
}
```

Si le nombre est divisible par 2 on ne teste pas inutilement s'il est divisible par 3.

Remarque

Le programme précédent peut aussi s'écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  i = 1;
  while(i <= 1000) {
    if(i%2 == 0 || i%3 == 0)
      cout << i << endl;
    i++;
  }
  return 0;
}
```

Comme dans le programme précédent, si `i%2==0` est vrai, le test `i%3==0`, inutile puisqu'il s'agit d'un « ou » , n'est pas effectué. La seconde version est donc meilleure puisque plus simple.

2.2 Boucle `do ... while`

2.2.1 Forme générale

Elle ressemble beaucoup à la précédente mais, contrairement à celle-là, elle est toujours exécutée au moins une fois. Sa forme générale est :

```
do {
  instructions;
} while(expression);
```

Les instructions du bloc sont tout d'abord exécutées une première fois, puis `expression` est testée :

si elle vraie (valeur 1) le bloc est de nouveau exécuté et ainsi de suite jusqu'à ce que `expression` devienne fausse

si elle est fausse (valeur 0) le programme se poursuit avec les instructions suivant le `while` et la boucle n'aura donc été exécutée qu'une seule fois.

2.2.2 Exemple

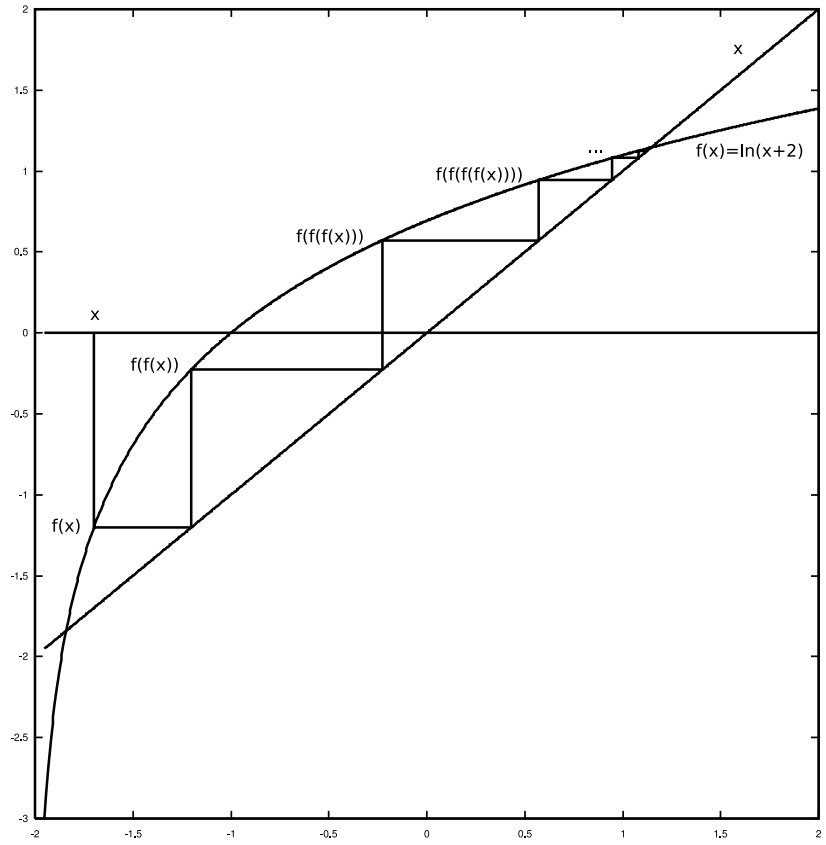


FIGURE 7 –

```
/* Resolution de l'equation x=log(x+2.) */
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  double x, eps, xa, diff;
  x = -1.75; eps = 1.e-5;
  do {
    xa = x;
    x = log(x+2.);
    diff = fabs(x-xa);
  } while(diff >= eps);
  cout << "Racine=" << x << endl;
  return 0;
}
```

Remarques

On voit ici l'utilité du `do ... while` par rapport au `while` : au début du premier tour de boucle la quantité `diff` sur laquelle est effectué le test n'est pas connue. Avec un `while` il faudrait l'initialiser à une valeur « artificielle» (par exemple `2*eps`).

Dans le programme précédent rien ne garantit à priori que `diff` va devenir inférieur à `eps`. Il peut donc être prudent d'ajouter une limite au nombre de tours de boucle effectués.

2.3 Boucle `for`

Elle permet d'imposer à l'avance le nombre de tours de boucle à effectuer.

2.3.1 Exemple

On veut calculer les 10 premières puissances de 2 :

```
x = 1;
for(i = 1; i <= 10; i++) {
  x = x*2;
}
```

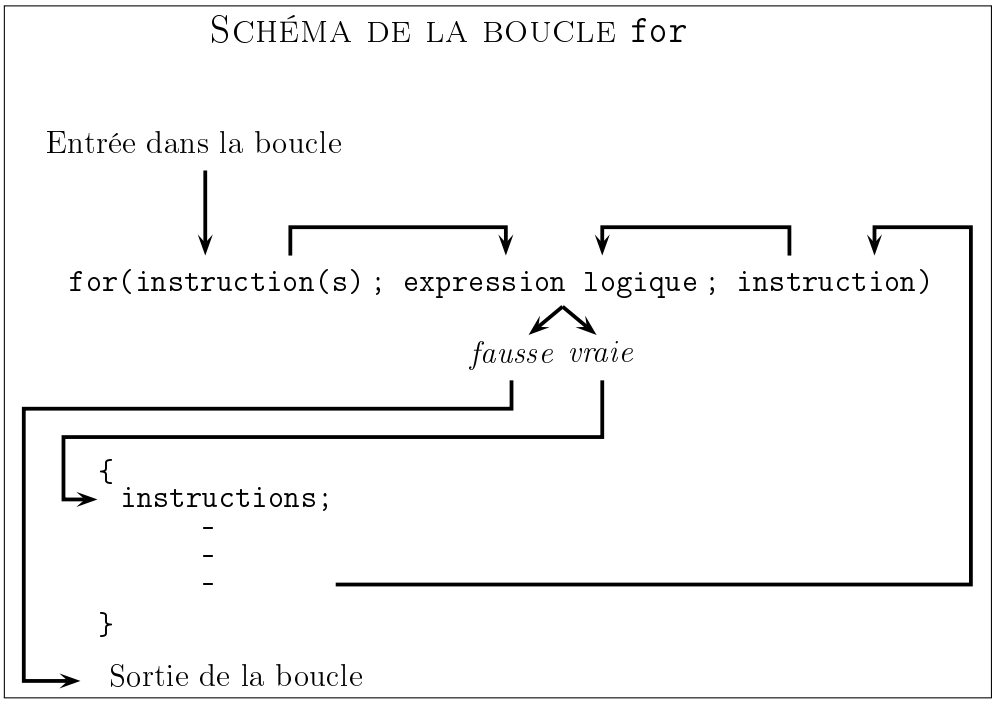
La boucle sera décrite pour `i` variant de 1 à 10 inclus.

2.3.2 Forme générale

Les parenthèses suivant le `for` contiennent trois champs séparés par deux point-virgules :

```
for(champ 1; champ 2; champ 3) {
  ...
}
```

Le `champ 1` contient une ou plusieurs instructions séparées par des virgules, exécutées une seule fois avant d'entrer dans la boucle (initialisation).
Le `champ 2` contient une expression à valeur logique calculée et interprétée comme « vrai» ou « faux» au début de chaque tour.
Le `champ 3` contient une instruction exécutée à la fin de chaque tour.



Chacun des trois champs est facultatif. Ainsi :

```
for(; expression logique;) {
  ...
}
```

est strictement équivalente à `while(expression logique)`.
Et :

```
for(;;) {
  ...
}
```

est une boucle infinie équivalente à `while(1)`, le `champ 2` vide étant considéré comme « vrai» . Pour en sortir il faut utiliser un `break`, étudié précédemment.

Remarque

L'initialisation peut contenir plusieurs instructions séparées par des virgules. Par exemple, dans le calcul des 10 premières puissances de 2 vu précédemment l'initialisation de `x` à 1. peut se faire dans le `for` :

```
for(x = 1., i = 1; i <= 10; i++) {
  x = x*2;
}
```

Remarque

Toute boucle `for` peut être écrite avec un `while` et inversement. On s'efforce de choisir la solution la plus simple et la plus lisible.

Exemples

1) Calcul des puissances de 2 jusqu'à 2^n.

```
#include<iostream>
using namespace std;
int main() {
  int i, i2, n;
  n = 10; i2 = 1;
  //-----------------------------------------------
  for(i = 0; i <= n; i++) {
    cout << "2 puissance " << i << " = " << i2 << endl;
    i2 *= 2;
  }
  //-----------------------------------------------
  return 0;
}
```

On obtient :
2 puissance 0 = 1
2 puissance 1 = 2
...
2 puissance 10 = 1024
Pour écrire le même programme avec un `while` on remplace ce qui est entre tirets dans le programme précédent par :

```
//-----------------------------------------------
i = 0;
while(i <= n) {
  cout << "2 puissance " << i << " = " << i2 << endl;
  i2 *= 2;
  i++;
}
//-----------------------------------------------
```

2) Calcul d'une intégrale par la méthode des rectangles.

$$\int_a^b f(x)\,dx \simeq h\sum_{i=1}^{n} f\left[a+(i-\frac{1}{2})h\right]$$

où n est le nombre de rectangles, h le pas d'intégration : $h = (b-a)/n$.
Appliquons cette formule à la fonction :

$$\frac{1}{1+x^2}$$

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  double a = 1., b = 2., s = 0., h, som, u;
  int i, n = 1000;
  h = (b-a)/n;
  for(i = 1; i <= n; i++) {
    u = a + (i-0.5)*h;
    s += 1./(1.+u*u);
  }
  som = h*s;
  cout << "Valeur par les rectangles : " << som << endl;
  cout << "Valeur par la primitive : " << atan(b)-atan(a) << endl;
  return 0;
}
```

On obtient :
Valeur par les rectangles : 0.321750
Valeur par la primitive : 0.321751

6. Entrées, sorties, fichiers, en C++

Les entrées sont les données que l'utilisateur fournit à un programme durant l'exécution, par l'intermédiaire du clavier ou d'un fichier.
Les sorties sont les résultats que le programme fournit à l'utilisateur durant l'exécution, par l'intermédiaire de l'écran ou d'un fichier.
On utilise les instructions d'entrée-sortie du C++ parce qu'elles sont plus simples que celles du C. Un chapitre, hors programme, est consacré à celles du C.
On n'indique ici que le strict minimum.
Il faut, dans tous les cas, ajouter au début du fichier contenant le programme :

```
#include<iostream>
using namespace std;
```

1 Afficher des résultats à l'écran

Pour afficher à l'écran la valeur d'une variable `x` :

```
cout << x << endl;
```

Le `endl` signifie que le programme devra aller à la ligne après la valeur de `x`.
Pour ajouter des commentaires, par exemple : *Valeur de x :*

```
cout << "Valeur de x : " << x << endl;
```

Pour afficher les valeurs de plusieurs variables `x`, `y` et `z` :

```
cout << x << " " << y << " " << z << endl;
```

Il faut au moins un commentaire blanc pour séparer les valeurs des variables.

2 Entrer des données à partir du clavier

Pour attribuer une valeur à une variable `x` à partir du clavier :

```
cin >> x;
```

Lors de l'exécution, la valeur entrée au clavier doit être validée par un *Entrée*.
Il faut faire précéder cette instruction d'une demande telle que :

```
cout << "Valeur de x ? ";
```

sinon le programme reste muet dans l'attente de la valeur et l'utilisateur ne comprend pas nécessairement que c'est à lui d'agir.
Pour lire plusieurs valeurs à la fois :

```
cout << "Valeur de x y z ? ";
cin >> x >> y >> z;
```

Les valeurs entrées au clavier doivent être séparées par des blancs ou des *Entrée* et la dernière (celle de `z` ici), suivie d'un *Entrée* de validation.
Tant qu'un programme n'est pas au point, il ne faut pas utiliser de `cin`, mais mettre les valeurs des données dans le programme sous la forme `x = 3.12`, etc. Sinon, à chaque exécution destinée à tester le programme, il faut taper les données au clavier, ce qui est une perte de temps. Ensuite, si le programme est destiné à être utilisé, on peut mettre des `cin`, mais à condition qu'ils soient en nombre très limité. Sinon, l'entrée des données au clavier est trop fastidieuse et comporte trop de risques d'erreur. Il vaut mieux placer d'abord les données dans un fichier qui leur est dédié et les faire lire par le programme dans ce fichier (voir ci-dessous l'emploi des fichiers).
Inversement, il n'est pas normal que l'utilisateur d'un programme considéré comme achevé soit contraint d'entrer ses données dans le fichier du programme lui-même.

3 Écrire dans un fichier

Pour écrire dans un fichier, il faut ajouter la directive :

```
#include<fstream>
```

et ouvrir le fichier en écriture par :

```
fstream fich;
fich.open("programme.res", ios::out);
```

Schématiquement on peut dire que le fichier dans lequel on veut écrire a deux noms, tous les deux choisis par l'utilisateur :

1. l'un pour *Linux* , dans cet exemple : *programme.res*
2. l'autre pour le programme C, dans cet exemple `fich`

Ensuite, on procède exactement de la même façon qu'avec l'affichage à l'écran, en remplaçant simplement partout `cout` par `fich`. Par exemple, pour écrire la valeur de la variable `x` dans le fichier connu de *Linux* sous le nom *programme.res* :

```
fich << x << endl;
```

Il faut fermer le fichier après la dernière instruction qui l'utilise par :

```
fich.close();
```

Les fichiers encore ouverts à la fin de l'exécution du programme sont fermés automatiquement.

Remarque

Tout comme la déclaration et l'initialisation d'une variable d'un autre type (comme par exemple un `int`), il est aussi possible de combiner la déclaration et l'initialisation d'une variable du type `fstream` dans une seule instruction. Ainsi on peut écrire `fstream fich("programme.res",ios::out);` au lieu de `fstream fich; fich.open("programme.res",ios::out);`

4 Lire dans un fichier

Pour lire dans un fichier, il faut, comme pour y écrire, ajouter la directive :

```
#include<fstream>
```

et ouvrir le fichier en lecture par :

```
fstream fich;
fich.open("programme.dat", ios::in);
```

Ensuite, on procède exactement de le même façon qu'avec l'entrée des données au clavier, en remplaçant simplement partout `cin` par `fich`. Par exemple, pour attribuer une valeur écrite dans le fichier connu de *Linux* sous le nom *programme.res*, à la variable `x` :

```
fich >> x;
```

4.1 Lire dans un fichier contenant un nombre de lignes connu à l'avance

On suppose qu'on a un fichier de nom *Linux coordonnees.dat*, contenant un triplet de valeurs par ligne, comme dans l'exemple suivant :

1.76 3.45 7.81
9.87 4.65 7.33
6.78 0.67 6.23
...

et qu'on sait que ce fichier contient `n` lignes. Ces valeurs peuvent être lues par les instructions :

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
  double x, y, z;
  int i, n = ...;
  fstream fich;
  fich.open("coordonnees.dat", ios::in);
  for(i = 1; i <= n; i++) {
    fich >> x >> y >> z;
    cout << x << " " << y << " " << z << endl; // vérification de la lecture
    ...
```

```
  }
  fich.close();
  return 0;
}
```

4.2 Lire dans un fichier contenant un nombre de lignes non connu à l'avance

La seule différence avec le paragraphe précédent est qu'on ne connaît pas à priori le nombre de lignes du fichier. La condition d'arrêt de la boucle de lecture est donnée par la valeur de l'instruction `fich >> x >> y >> z`.

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
  double x, y, z;
  int i;
  fstream fich;
  fich.open("coordonnees.dat", ios::in);
  i = 0;
  while(fich >> x >> y >> z) {
    cout << x << " " << y << " " << z << endl; // vérification de la lecture
    i++;                                       // comptage du nombre de lignes
    ...
  }
  cout << "Le fichier coordonnees.dat a " << i << " lignes" << endl;
  fich.close();
  return 0;
}
```

5 Fermer puis ré-ouvrir un fichier dans un même programme pour y lire ou écrire

5.1 Fermer puis ré-ouvrir et lire

Le programme suivant écrit dans le fichier *toto.txt*, le ferme, puis le ré-ouvre et y lit à partir du début :

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
  int i = 1;
  fstream fich;
  fich.open("toto.txt", ios::out);
  fich << i+1 << endl;
  fich.close();
  fich.open("toto.txt", ios::in);
  fich >> i;
  fich.close();
  cout << "i=" << i << endl;
  return 0;
}
```

5.2 Fermer puis ré-ouvrir et écrire

Le programme suivant écrit dans le fichier *toto.txt*, le ferme, puis le ré-ouvre et y écrit à la suite :

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
```

```
  int i = 1;
  fstream fich;
  fich.open("toto.txt", ios::out);
  fich << i << endl;
  fich.close();
  fich.open("toto.txt", ios::out|ios::app);
  fich << i+1 << endl;
  fich.close();
  return 0;
}
```

Si, au lieu d'écrire à la suite, on veut écrire au début, donc effacer ce qui était précédemment écrit, on remplace la ligne :

```
fich.open("toto.txt", ios::out|ios::app);
```

par :

```
fich.open("toto.txt", ios::out|ios::trunc);
```

ou en fait par juste `ios::out` parce que `ios::trunc` est le mode par défaut.

6 Passer un nom de fichier en argument d'une fonction

Exemple :

```
...
int f(..., fstream &fich, ...) {
  ...
  fich << ... << endl;
  ...
  return 0;
}
int main() {
  ...
  fstream res("mon_fichier.res", ios::out);
  f(...,res,...);
  ...
  return 0;
}
```

C'est un passage par référence, propre au C++.

7 Présentation de l'affichage à l'écran et de l'écriture dans un fichier

Pour ce qui suit il faut ajouter la directive :

```
#include<iomanip>
```

7.1 Notation standard, fixe ou scientifique

```
cout << x << endl; // affiche en notation standard
cout << fixed; // passe en notation fixe pour les cout suivants
cout << scientific; // passe en notation scientifique pour les cout suivants
cout.setf(ios_base::floatfield); // revient en notation standard
```

7.2 Précision (nombre de chiffres affichés)

```
n = cout.precision(); // stocke la précision actuelle dans la variable entière n
cout << setprecision(15); // fixe la précision à 15 pour les cout suivants
cout << setprecision(n); // revient à la précision initiale
```

7.3 Largeur minimum consacrée à chaque affichage

```
cout << setw(10) << x << endl; // impose une largeur minimum, n'agit que sur
                               // l'élément suivant (largeur nulle par défaut)
```

Si on écrit dans un fichier au lieu d'afficher à l'écran, toutes ces instructions restent valides, il suffit de remplacer partout `cout` par `fich`.

7.4 Exemple récapitulatif

Le programme suivant :

```
#include<iostream>
#include<iomanip>
#include<fstream>
using namespace std;
int main() {
  double x = 12345.6789012345;
  int n;
// Notation standard, fixe ou scientifique ------------------------------------------------------
  cout << "x=" << x << endl; // affiche en notation standard
  cout << fixed; // passe en notation fixe pour les cout suivants
  cout << "x=" << x << endl;
  cout << scientific; // passe en notation scientifique pour les cout suivants
  cout << "x=" << x << endl;
  cout.setf(ios_base::floatfield); // revient en notation standard
  cout << "x=" << x << endl << endl;
// Précision (nombre de chiffres affichés) ------------------------------------------------------
  n = cout.precision(); // stocke la précision actuelle dans la variable entière n
  cout << "La précision actuelle est : " << n << endl;
  cout << "x=" << x << endl;
  cout << setprecision(15); // fixe la précision à 15 pour les cout suivants
  cout << "x=" << x << endl;
  cout << setprecision(n); // revient à la précision initiale
  cout << "x=" << x << endl;
// Largeur minimum consacrée à chaque affichage -------------------------------------------------
  cout << "x=" << setw(10) << x << endl; // impose une largeur minimum, n'agit que sur
                                         // l'élément suivant (largeur nulle par défaut)
  return 0;
}
```

donne le résultat :

x=12345.7
x=12345.678901
x=1.234568e+04
x=12345.7

La précision actuelle est : 6
x=12345.7
x=12345.6789012345

x=12345.7
x= 12345.7

8 Créer une séquence de fichiers

8.1 Méthode C

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

```
#include<stdlib.h>
#include<iostream>
#include<fstream>
using namespace std;
int main() {
  char* s = (char*)malloc(50*sizeof(char)); // voir chapitre Tableaux dynamiques
  int i, n = 13;
  fstream fich;
  for(i = 0; i < n; i++) {
    // %03d pour avoir une séquence 000 001 002 etc.
    sprintf(s, "%s%03d%s", "fichier_", i, ".dat");
    cout << s << endl;
    fich.open(s, ios::out);
    fich << "Contenu du fichier" << " " << i << endl;
    fich.close();
  }
  return 0;
}
```

8.2 Méthode C++

```
// On ne peut pas écrire directement dans une string, il faut passer par un ostringstream.
#include<iostream>
#include<iomanip>
#include<fstream>
#include<stdlib.h>
#include<sstream>
#include<string>
using namespace std;
int main() {
  ostringstream y;
  int n = 13, i;
  fstream fich;
  for(i = 0; i < n; i++) {
    // setfill et setw pour avoir une séquence 000 001 002 etc.
    y << "fichier_" << setfill('0') << setw(3) << i << ".dat";
    fich.open((y.str()).c_str(), ios::out);
    fich << "Contenu du fichier " << i << endl;
    fich.close();
    y.str(""); // vide l'ostringstream
  }
  return 0;
}
```

9 Redirections d'entrée et de sortie

Il existe une autre possibilité pour écrire et lire dans un fichier : la « redirection » du résultat d'une commande *Linux*. Si on fait compiler et exécuter le programme contenu dans le fichier *jojo.cpp* par :

$ ccc jojo.cpp > lili

au lieu de simplement :

$ ccc jojo.cpp

tout ce qui aurait dû s'afficher à l'écran[23] est envoyé dans le fichier de nom *Linux lili*.

Remarque :

23. Donc toutes les sorties produites par `cout`.

Dans le cas où le fichier *lili* existe déjà :

$ ccc jojo.cpp >> lili

ajoute les sorties de `cout` au fichier *lili* sans effacer son contenu initial.

De même :

$ ccc jojo.cpp < dudu

tout ce qui aurait dû être entré au clavier[24] est envoyé par le fichier *dudu*.
Il faut bien remarquer qu'ici, c'est la commande *Linux* qui provoque l'écriture ou la lecture dans un fichier et non une instruction du C. C'est une méthode simple et efficace pour enregistrer des résultats ou fournir des données.

10 Remarque pour amateurs avertis

En plus de faire un retour à la ligne en ajoutant un caractère `\n`, `endl` purge le buffer de sortie et force ainsi son écriture en appelant `ostream::flush()` (cela a le même fonctionnement que la fonction `fflush()` du C). Les deux lignes de code suivantes sont donc équivalentes :

```
cout << "coucou" << endl;
cout << "coucou\n" << flush;
```

Il faut donc être prudent avec son utilisation, notamment avec les fichiers, car une opération de flush n'est pas gratuite. Son utilisation fréquente peut même sérieusement grever les performances en annulant tous les bénéfices d'une écriture bufférisée (*http ://cpp.developpez.com/faq/cpp/index.php ?page=SL#SL_ endl*).

24. Donc toutes les entrées produites par `cin`.

7. Fonctions

1 Définition et utilité des fonctions

Les fonctions sont des parties de programme totalement ou partiellement indépendantes du reste, auxquelles on peut fournir des données, et qui accomplissent une tâche en renvoyant éventuellement un résultat.

L'indépendance des fonctions fait leur intérêt : on peut les utiliser, dans une certaine mesure, comme des boîtes noires, sans avoir à se soucier de leur intégration dans le programme qui les utilise. Ceci clarifie grandement la construction et le développement des programmes.

Dans ce chapitre on considérera uniquement des fonctions écrites dans le même fichier [25] que le programme principal. Pour une fonction d'usage universel, par exemple la multiplication de matrices, ce n'est pas pratique puisqu'il faut recopier cette fonction dans le fichier de chaque programme qui y fait appel. Dans un chapitre ultérieur on verra comment mettre des fonctions dans des fichiers autonomes tout en pouvant les utiliser à partir d'autres fichiers. Ceci permet de constituer des « bibliothèques » de fonctions écrites une fois pour toutes.

2 Exemples

Exemple 1

Supposons qu'on ait à calculer la quantité :

$$y = \frac{1}{1+x^2}$$

pour diverses valeurs de x : x_1, x_2, ... Au lieu d'écrire :

```
y1 = 1./(1.+x1*x1)
y2 = 1./(1.+x2*x2)
...
```

il est plus élégant et plus lisible de définir une fonction, au sens de la programmation, nommée par exemple `y`, qui calcule y pour toute valeur de x fournie en argument. Ceci s'écrit :

```
#include<iostream>
using namespace std;
int main() {
  double x1, x2;
  double y(double);    // <- prototype
  x1 = 1.; x2 = 4.;
  cout << y(x1) << endl;
  cout << y(x2) << endl;
  return 0;
}
double y(double x) {   // <- en-tete
  return 1./(1.+x*x);
}
```

On voit que dans ce cas il existe un autre bloc que celui de `main`. C'est dans ce bloc supplémentaire qu'est définie la fonction. Le programme est donc divisé en deux parties :
— le `main`
— une fonction comprise dans le bloc suivant l'instruction `double y(double x)` (nommée « en-tête de la fonction »)
On remarque qu'il faut déclarer le type de la fonction et de ses arguments deux fois :
— une fois à l'intérieur de `main` par `double y(double)` (cette déclaration se nomme un « prototype »)
— une fois dans l'en-tête de la fonction par `double y(double x)`
On voit que dans un prototype il n'est pas nécessaire d'inclure les noms des arguments (ici `x`), seulement leurs types (ici `double`). Par contre, ce n'est pas proscrit non plus de les inclure et cela peut rendre un prototype plus lisible.

Remarque

25. On peut dire aussi, ce qui est équivalent, dans le même module.

Dans cet exemple :

`x` est la variable de la fonction, on l'appelle « argument muet »
`x1`, `x2`, ... sont des valeurs particulières de `x` pour lesquelles on veut calculer la fonction, on les appelle « arguments effectifs » .

Remarque importante

L'ordre dans lequel on écrit le `main` et la fonction `y` n'est pas imposé. Si on écrit la fonction en premier il n'est plus nécessaire de la déclarer dans le `main`. Cette possibilité est intéressante si on veut éviter des déclarations redondantes.

Exemple 2

Calcul de la distance de deux points du plan avec une fonction. La différence avec l'exemple précédent est qu'il y a plusieurs arguments et qu'on a défini en premier la fonction.

```
#include<iostream>
#include<math.h>
using namespace std;
double dist(double xa, double ya, double xb, double yb) { // <- en-tete de la fonct. dist
  return sqrt((xb-xa)*(xb-xa) + (yb-ya)*(yb-ya));
}
int main() {
  double dist(double, double, double, double); // <- prototype facultatif ici
                                               // parceque la définition de
                                               // dist précède la fonction main
                                               // qui l'appelle
  double x1, y1, x2, y2;
  x1 = 1.; y1 = 2.; x2 = 3.; y2 = 4.;
  cout << dist(x1,y1,x2,y2) << endl;
  x1 = 0.5; y1 = -1.; x2 = 3.7; y2 = -0.3;
  cout << dist(x1,y1,x2,y2) << endl;
  return 0;
}
```

Exemple 3

Fonction sinus cardinal $\sin x/x$, en prévoyant le cas $x = 0$:

```
#include<iostream>
#include <math.h>
using namespace std;
double sinc(double x) {
  if (x == 0.)
    return 1;
  else
    return sin(x)/x;
}
int main() {
  double u;
  u = 1.5;
  cout << sinc(u) << endl;
  return 0;
}
```

Une fonction peut ne pas retourner de résultat ou ne pas avoir d'argument.

— s'il n'y a pas de résultat retourné : dans l'en-tête on remplace le type de la fonction par le mot clé `void`
— s'il n'y a pas d'argument : dans les parenthèses de l'en-tête on remplace les types et les noms des arguments par rien du tout (parenthèses vides).

Deux exemples de telles fonctions :

Exemple 4

Fonction imprimant les n premières puissances de 2 (de 2^0 à 2^{n-1}) et un exemple de son appel avec n=10 :

```
#include<iostream>
using namespace std;
void p(int n) {
  int i, x = 1;
  for (i = 0; i <= n-1; i++) {
    cout << x << endl;
    x = x*2;
  }
}
int main() {
  p(10);
  return 0;
}
```

Exemple 5

Fonction qui, à chaque appel, écrit une ligne d'étoiles :

```
#include<iostream>
using namespace std;
void ligne_etoiles() {
  cout << "****************" << endl;
}
int main() {
  ligne_etoiles();
  return 0;
}
```

Exemple 6

On a vu au chapitre **Tests et boucles** un programme calculant, par la méthode des rectangles, l'intégrale :

$$\int_a^b \frac{1}{1+x^2}\,dx$$

Réorganisons le en écrivant séparément la fonction à intégrer, le calcul de l'intégrale et le programme principal :

```
#include<iostream>
using namespace std;
double f(double x) {
  return 1./(1.+x*x);
}
double integ(double a,double b,int n) {
  double h, s;
  int i;
  h = (b-a)/n; s = 0.;
  for (i = 1; i <= n; i++)
    s = s + f(a+(i-0.5)*h);
  return h*s;
}
int main() {
  double x1, x2;
  int np;
  x1 = -0.57; x2 = 1.17; np = 1000;
  cout << integ(x1,x2,np) << endl;
  return 0;
}
```

Sous cette forme le programme est plus lisible, il est plus facile de changer la fonction à intégrer sans risquer d'erreur, il suffit pour cela de modifier l'expression de `f` dans sa définition.

Remarque

On voit sur ces exemples qu'une fonction peut à la fois effectuer une tâche et renvoyer une valeur.

3 Passage des valeurs en argument

Il est important de comprendre comment les valeurs des arguments sont transmises à une fonction. Une fonction est un élément de programme qui s'exécute à chaque appel. Si, par exemple, on a une fonction d'en-tête :

```
int f(int x)
  ...
```

et qu'on l'appelle par :

```
  ...
  y=3;
  ... f(y);
```

ou :

```
  ...
  ... f(3);
```

cela signifie que la fonction s'exécute comme si on avait ajouté à son début l'instruction `x=3` et c'est tout. On appelle ce type de transmission des valeurs des arguments « passage par valeur » . Supposons que la fonction contienne une instruction qui modifie la valeur de `x` fournie par l'appel (`3` dans ce cas) comme suit :

```
int f(int x)
  ...
  x = 4
  ...
```

Lorsque l'exécution de la fonction est terminée et que l'on retourne à la fonction appelante (ici `main`), la valeur de la variable effective `y` du programme appelant est inchangée, elle est restée égale à `3`. La transmission des valeurs se fait uniquement dans le sens fonction appelante vers la fonction appelée. Ce fonctionnement est différent de celui d'un langage comme le Fortran où la transmission peut avoir lieu dans les deux sens et est dite « par adresse » . [26]

4 Variables locales, variables globales

Les variables déclarées dans une fonction (variables déclarées à l'intérieur et arguments muets) sont dites « locales » à la fonction (ceci s'applique au `main` qui peut être considéré comme une fonction particulière sans argument). Les variables de l'une des fonctions ne sont pas accessibles à l'autre et réciproquement. Les variables de deux fonctions différentes n'ont rien à voir entre elles : même si elles portent le même nom elles désignent des emplacements de mémoire différents, leurs contenus sont complètement indépendants. C'est d'ailleurs ce qui fait l'intérêt principal des fonctions : on peut en écrire une sans se soucier si les noms de variables utilisés sont déjà employés ailleurs dans le programme, ce qui assure automatiquement son indépendance.
Cependant il est souvent nécessaire de disposer de variables communes à deux ou plusieurs fonctions. Si, par exemple, un ensemble de paramètres est utilisé par plusieurs fonctions il est compliqué de devoir soit les définir dans une des fonctions et les passer en argument aux autres soit les définir dans chacune des fonctions. La solution est donnée par l'emploi de variables « globales » . On peut déclarer des variables à l'extérieur de toute fonction : elles sont alors communes à toutes les fonctions suivant cette déclaration. Une modification de leur contenu est connue simultanément de toutes les fonctions qui les ont en commun. Considérons un programme contenant une fonction nommée `f` :

```
int main() {
  double x,z;
  ...
}
... f(...) {          /* symbolise l'en tete de la fonction f */
  double y,z;
  ...
}
```

`x` est inconnue de la fonction, `y` est inconnue de `main`. Les variables `z` de `main` et de la fonction n'ont rien à voir entre elles (elles pourraient très bien ne pas être du même type, par exemple).
Si maintenant on déclare comme suit :

26. On verra que la transmission peut aussi se faire par adresse en C, au chapitre **Pointeurs**.

```
double z;
int main() {
  double x;
  ...
}
... f(...) {
  double y;
  ...
}
```

Les statuts de `x` et `y` sont inchangés mais `z` devient unique et commune au `main` et à la fonction.
Si on veut qu'une variable `v` soit commune uniquement à deux fonctions `f1` et `f2`, il faut écrire :

```
int main() {
  ...
}
double v;
... f1(...) {
  ...
}
... f2(...) {
  ...
}
```

car `v` n'est commune qu'aux fonctions qui suivent sa déclaration. La définition des fonctions `f1` et `f2` se trouvant dans ce cas après `main`, il est nécessaire de déclarer `f1` et `f2` par leur prototype dans `main`.

Remarque

En cas d'homonymie, dans une fonction, entre une variable locale et une variable globale c'est la variable locale qui est prise en considération, on dit que la variable globale est « masquée » .

5 Durée de vie des variables

Les variables locales des fonctions n'existent que le temps de l'exécution de la fonction. Au retour à la fonction appelante les emplacements mémoire de ces variables sont libérés pour être utilisés à autre chose. Si on a donné une certaine valeur à une variable locale d'une fonction lors d'un premier appel, cette valeur ne sera pas retrouvée à l'appel suivant. Pour la fonction `main` la question ne se pose pas puisqu'elle n'est exécutée qu'une fois.
Si l'on a besoin que la valeur de la variable soit conservée d'un appel à l'autre il faut la déclarer avec l'attribut `static` :

```
static int i;
```

par exemple. Une telle variable est dite « statique » . Cela ne veut pas dire que `i` est constante mais qu'elle a, au début d'un appel, la valeur qu'elle avait à la fin de l'appel précédent. Un exemple sera vu à la section **Initialisation des variables**.
Une variable locale non statique est dite « automatique » .

6 Initialisation des variables

6.1 Variables locales du programme principal

Les variables déclarées dans le `main` ne sont pas initialisées à priori, leur valeur peut être n'importe quoi tant que l'utilisateur ne leur en a pas attribué explicitement une, ce qui peut se faire par :

```
int i = 1;
```

ou, ce qui est équivalent, par :

```
int i;
i = 1;
```

sauf s'il s'agit d'une constante qui doit nécessairement être initialisée et ne peut l'être que par :

```
const int i = 1;
```

6.2 Variables locales des fonctions

6.2.1 Cas des variables automatiques

Elles ne sont pas initialisées à priori. Elles peuvent l'être par :

```
int i;
i = 1;
```

ou

```
int i = 1;
```

et dans les deux cas l'initialisation est faite à chaque appel de la fonction.

6.2.2 Cas des variables statiques

Elles sont initialisées à priori à 0. On peut les initialiser à une autre valeur par :

```
static int i = 1;
```

et dans les deux cas l'initialisation n'est faite qu'au premier appel, ce qui est évidemment nécessaire.

6.3 Variables globales

Elles sont à priori initialisées à 0 par le compilateur, on peut bien sûr les initialiser à une autre valeur par :

```
int j = 1;
```

6.4 Valeurs autorisées pour l'initialisation

6.4.1 Variables statiques

Elles ne peuvent être initialisées que par des valeurs connues à la compilation, c'est à dire par des constantes explicites ou déclarées par `const`, ou encore des expressions constituées de telles constantes, comme dans l'exemple suivant :

```
const int m = 3, n = 5;
static int k = m*n + 2*m + 4*n;
...
```

6.4.2 Variables automatiques

Elles peuvent être initialisées par une expression contenant des variables et des fonctions quelconques ayant été définies précédemment :

```
int m = 3, n = 5, k = m*n + 2*m + 4*n;
double pi = acos(-1.), dpi = 2*pi;
```

car ces dernières déclarations ne sont qu'une abréviation de :

```
int m, n, k;
double pi, dpi;
m = 3; n = 5; k = m*n + 2*m + 4*n;
pi = acos(-1.); dpi = 2*pi;
```

7 Appel d'une fonction par une fonction

Toute fonction peut être appelée par n'importe quelle autre fonction, à condition que la fonction appelée soit déclarée par son prototype dans la fonction appelante ou qu'elle soit placée avant la fonction appelante dans le fichier contenant les instructions du programme. L'argument d'une fonction peut très bien être la valeur retournée par elle même ou une autre fonction donc on peut écrire :

`f(f(f(...)))` ou `f(g(h(...)))` sans limitation [27]

27. Fonction composée, à ne pas confondre avec la notion de fonction récursive.

8 Fonctions à nombre variable d'arguments

Une fonction peut avoir un nombre variable d'arguments. Considérons la fonction suivante, nommée `prod`, à laquelle on fournit en arguments d'entrée :

- un entier par l'intermédiaire de la variable `n`
- `n` nombres

et dont la valeur est le produit des `n` nombres.

```
#include<iostream>
#include<stdarg.h>
using namespace std;
double prod(int n, ...) {
  int i; double p;
  va_list ap;
  va_start(ap,n);
  for(p = 1, i = 0; i < n; i++)
    p *= va_arg(ap,double);
  va_end(ap);
  return p;
}
/*************************************************************************/
int main() {
  cout << prod(2, 1.5, 3.) << endl;
  cout << prod(3, 1.2, 4., 10.) << endl;
  return 0;
}
```

Les pointillés dans l'en-tête de la fonction signifient que c'est une fonction à nombre variable d'arguments.[28] Une telle fonction doit avoir au moins un argument fixe, ici c'est `n`, et le ou les arguments fixes doivent figurer en premier. Le type `va_list` et les fonctions pré-définies `va_start`, `va_arg`, `va_end` font partie de la bibliothèque `stdarg.h`, il faut donc la directive `#include<stdarg.h>`.
Le second argument de `va_start` (ici `n`) doit être le dernier des arguments muets fixes de la fonction à nombre variable d'arguments. Ce n'est pas nécessairement une variable contenant le nombre d'arguments variables effectifs dans l'appel.
Il n'est pas demandé aux étudiants de maîtriser les fonctions de ce type, mais la connaissance de leur existence et de leur utilisation peut être utile en pratique. Elles ne seront employées dans ce cours que pour initialiser les tableaux dynamiques, étudiés dans la suite.

9 Fonctions récursives

Ce sont des fonctions qui s'utilisent elles-même dans leur définition, ce qui est autorisé par le C. On peut par exemple écrire une fonction calculant $n!$ de la façon suivante :

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n*fac(n-1);
}
```

La récursivité conduit dans certains cas à une programmation élégante. Par contre elle peut conduire à un fort encombrement de la mémoire et à des calculs lents. Elle est mentionnée ici uniquement pour son importance « culturelle » et ne sera pas utilisée dans ce cours.

28. Contrairement à tous les autres cas dans ce poly, où les pointillés indiquent simplement des types, des arguments ou des lignes qu'on n'a pas explicités, ici il faut vraiment écrire des pointillés dans le programme.

10 Limites de l'utilisation des fonctions telle qu'elle vient d'être exposée

Un inconvénient majeur, dans le cadre étudié jusqu'à présent, est que les fonctions ne peuvent retourner que la valeur d'une seule variable. On ne peut pas écrire par exemple :

```
...
return (a,b,c);
```

On ne peut pas retourner par la valeur de la fonction les deux racines d'une équation du second degré ou les composantes d'un vecteur.
Il serait également très intéressant de pouvoir fournir une fonction en argument d'une fonction. Supposons par exemple que l'on écrive une fonction C, nommée `som`, par exemple, qui calcule numériquement l'intégrale :

$$\int_a^b f(x)\,dx$$

f étant une fonction fixée, par exemple $\dfrac{1}{1+x^2}$. La fonction `som` a pour argument `a` et `b`, on l'appelle donc par `som(a,b)`. Mais elle ne peut calculer l'intégrale que de la fonction $\dfrac{1}{1+x^2}$ puis celle-ci est incluse dans sa définition.
Il serait beaucoup plus général de pouvoir fournir aussi en argument la fonction à intégrer par un appel du genre `som(a,b,f)`, `f` étant la fonction à intégrer.
Pour parvenir à ces buts il faut utiliser les pointeurs.

8. Adresses, pointeurs

Remarque préliminaire

Dans ce chapitre, comme dans le reste du cours, on suppose que :

le type `int` occupe 4 octets, `double` 8 et `char` 1.

Il en est ainsi sur les PC utilisés en TD mais cela ne fait pas partie de la norme et peut varier d'un ordinateur à l'autre. Tout ce qui est exposé ici se transpose immédiatement aux autres cas.

1 Introduction

Dans ce cours les pointeurs sont présentés en vue de :

créer des variables indicées analogues à celles que l'on utilise en mathématiques (x_i, y_{ij}, z_{ijk}, etc.)
étendre les possibilités des fonctions (étudiées au chapitre **Fonctions**).

Le début de l'exposé sur les pointeurs est un peu formel, leur utilité n'apparaît qu'à partir de la section Application des pointeurs : transmission à une fonction par adresse.

2 Adresses

Dans la mémoire de l'ordinateur les octets sont numérotés, comme les chambres d'un hôtel. Quand on déclare une variable par son type et son nom on lui réserve une suite de n octets consécutifs : 4 pour un entier de type `int`, 8 pour un réel de type `double`, 1 pour un caractère de type `char`. Chaque variable a une « adresse » qui est le numéro du premier des octets qui lui sont réservés. La figure suivante schématise cette organisation pour les trois types précédents.

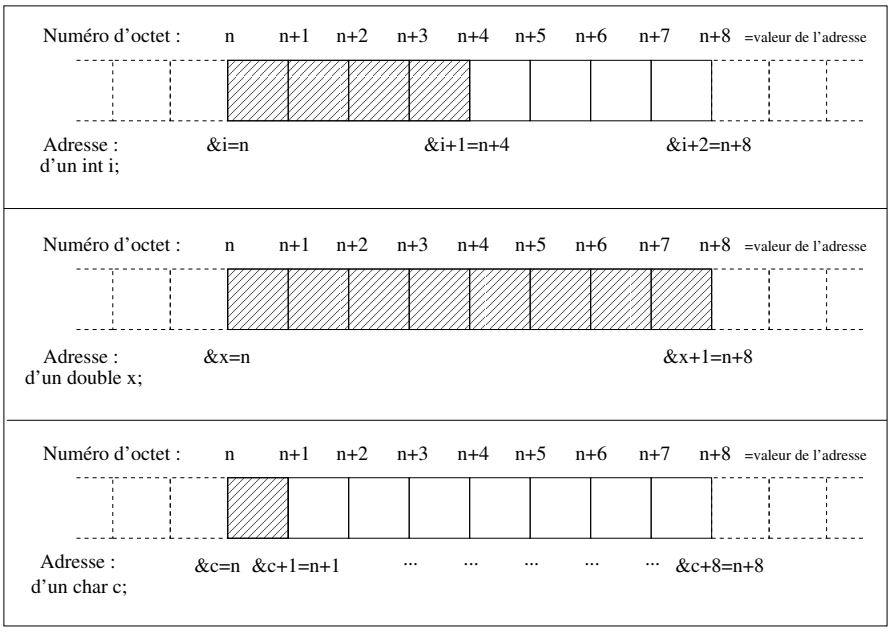


FIGURE 8 – Chaque carré symbolise un octet

L'adresse de la variable `i` est désignée par `&i`. Il est possible de connaître cette adresse explicitement par une instruction telle que `cout << (unsigned int)&i` [29], qui donne par exemple le résultat :

3219020668

Il faut bien distinguer entre l'adresse symbolique `&i` et sa valeur 3219020668. En effet si on écrit :

29. Si on écrivait simplement `cout << &i;` on obtiendrait cette adresse en notation hexadécimale.

```
#include<iostream>
using namespace std;
int main() {
  int i; double x;
  cout << (unsigned int)&i << " " << (unsigned int)(&i+1) << endl;
  cout << (unsigned int)&x << " " << (unsigned int)(&x+1) << endl;
  return 0;
}
```

on obtient :

```
3219020668 3219020672
3219020656 3219020664
```

ce qui montre que la valeur de l'adresse `&i+1` est égale à la valeur de l'adresse de `i` augmentée du nombre d'octets occupés par un `int` (c'est à dire 4) et que, de même, la valeur de l'adresse `&x+1` est égale à la valeur de l'adresse de `x` augmentée du nombre d'octets occupés par un `double` (c'est à dire 8).
Il faut remarquer également qu'on ne sait pas si ce qui se trouve à l'adresse `&i+1` est un `int` ou un `double` ou autre chose. Cette remarque s'applique aussi à `&x+1` sauf que, dans ce cas particulier, on peut affirmer qu'il ne s'agit pas d'un `double` puisqu'il n'y a pas assez de place pour un `double` à l'adresse 3219020664, l'octet à l'adresse 3219020668 étant déjà occupé par l'entier `i`. Les octets d'adresses 3219020664 à 3219020667 peuvent avoir été affectés à un `int`, des `char` ou autre chose, ou bien encore être libres, c'est à dire ne pas avoir été réservés.
De façon plus générale, `k` étant une variable entière, `&i+k` est l'adresse de `i` augmentée de `k` fois la taille d'un `int` et `&x+k` est l'adresse de `x` augmentée de `k` fois la taille d'un `double`. Mais, encore une fois, on ne sait pas à quoi sont affectés les octets suivant l'adresse `&i+k` ni ceux suivant l'adresse `&x+k` (sauf, évidemment, dans le cas particulier `k=0`).

On va voir dans ce qui suit qu'en général, on utilise les adresses par l'intermédiaire de variables d'un genre nouveau nommées « pointeurs » .

3 Pointeurs

3.1 Définition

Un pointeur est une variable qui contient non pas un nombre ou un caractère comme les variables étudiées jusqu'ici que nous pouvons qualifier d' « ordinaires » , mais l'adresse d'une variable « ordinaire » . Pour utiliser un pointeur il faut le déclarer en précisant le type de variable dont il pourra contenir l'adresse. Par exemple les instructions suivantes :

```
int* a;
double* b;
char* c;
```

déclarent trois pointeurs :

un nommé `a` susceptible de contenir une adresse de variable de type `int`
un nommé `b` susceptible de contenir une adresse de variable de type `double`
un nommé `c` susceptible de contenir une adresse de variable de type `char`.

Remarques

1. Un pointeur donné ne peut donc pas contenir l'adresse de n'importe quelle variable.
2. C'est le signe `*` qui indique que c'est une variable pointeur que l'on déclare.
3. Les déclarations `int* a;` et `int *a;` sont équivalentes pour le compilateur. La notation `int* a;` est plus claire : on déclare une variable qui s'appelle `a` et qui est du type « pointeur sur un `int` » (noté comme `int*`) et non pas une variable qui s'appelle `*a` et qui est du type `int`. Cependant, un peu illogique, quand on déclare plusieurs pointeurs avec une seule commande il faut obligatoirement répéter le signe `*` :

   ```
   int *a, *b;
   ```

Si on a déclaré les variables ordinaires suivantes :

```
int i;
double x;
char l;
```

on peut alors écrire des égalités telles que :

```
a = &i;
b = &x;
c = &l;
```

et `a` contient alors l'adresse de la variable entière `i`, `b` contient celle de la variable réelle `x`, `c` contient celle de la variable caractère `l`.

Remarque

L'attribution des adresses échappe complètement à l'utilisateur. Il est impossible d'écrire par exemple :

```
&x = ...
```

Une adresse ne peut donc pas figurer à gauche d'un signe `=`, on dit que ce n'est pas une « lvalue (left value) » , alors qu'un pointeur, lui, est une lvalue.

3.2 Opérations sur les pointeurs

3.2.1 L'opérateur *

L'opérateur `*` est un opérateur [30] qui s'applique à un pointeur : `*a` désigne la variable ordinaire dont l'adresse est celle contenue dans le pointeur `a`. Donc si `a` contient l'adresse de `i` [31], la variable `i` peut être désignée de deux façons différentes :

par `i` (façon habituelle)
par `*a` (par l'intermédiaire du pointeur).

Les instructions suivantes :

```
double i, j;
double* a;
...
a = &i;
j = *a;
```

attribuent à `j` la même valeur que si on avait simplement écrit :

```
double i, j;
...
j = i;
```

De même :

```
*a = j;
```

est équivalent à :

```
i = j;
```

Remarques

1. L'opérateur `*` s'applique aussi à une adresse : `*(&i)` est la même variable que `i`.
2. Suivant le contexte, l'opérateur `*` a deux significations en quelque sorte inverses l'une de l'autre :
   — la déclaration `int* i;` ou `int *i;` signifie que `i` est un pointeur (il faut voir le signe `*` ici comme faisant partie du nom du type `int*` et non pas comme un opérateur agissant sur `i`) ;
   — `i` étant un pointeur `*i` est une variable ordinaire.

3.2.2 Addition et soustraction d'entiers à des pointeurs, soustraction de pointeurs

À un pointeur on peut ajouter ou soustraire un entier et soustraire un autre pointeur du même type. Supposons que :

```
...
int k;
double x, y;
double *a, *b;
k = 5;
a = &x; b = &y;
...
```

30. Sans rapport avec la multiplication.
31. On dit alors que « `a` pointe sur `i` » .

donc que `a` pointe sur `x` et `b` sur `y`. Alors :

`a+k` est une adresse dont la valeur est égale à celle contenue dans le pointeur `a`, augmentée de `k` fois la taille d'un `double`, c'est donc ici `&x+k`[32]
`a-b` n'est pas une adresse, c'est simplement l'entier égal à la différence des valeurs de `&x` et `&y`.

Comme pour une adresse, une augmentation de `1` du pointeur augmente sa valeur d'un nombre d'octets qui dépend arithmétiquement du type du pointeur :

1 pour un `char`
4 pour un `int`
8 pour un `double`

Cette signification particulière des signes + et − pour les adresses et les pointeurs explique en partie pourquoi, en général, les pointeurs ont un type.

Remarque

`a+k` est une adresse et non un pointeur, on ne peut écrire par exemple `a+k = &x`.

Remarque importante

En conséquence de ce qui précède, `*(a+k)` est une variable ordinaire qui donne accès à ce qui est écrit à l'adresse de `x` augmentée de `k` fois la taille d'un `double`. Mais rien ne permet de savoir quel usage l'ordinateur fait de l'emplacement de mémoire situé à cette adresse[33]. Une instruction du type :

```
cout << *(a+k) << endl;
```

affichera ce qui est à l'adresse `&x+k` interprété comme un `double` mais ce n'en est pas un en général et, même si c'en est un, on ne sait pas à quoi il correspond.
Plus grave, une instruction du type :

```
*(a+k) = 1.;
```

par exemple, écrit à l'adresse `&x+k` alors que cet emplacement n'a pas été réservé et qu'il est affecté à un autre usage par l'ordinateur. Dans ce cas il y a violation de mémoire, ce qui peut entraîner des conséquences catastrophiques pour l'exécution du programme.
On ne se soucie pas de ce problème pour l'instant et on considère les manipulations sur les pointeurs de façon purement formelle. L'intérêt pratique de ces manipulations apparaîtra au chapitre **Tableaux dynamiques**.

3.2.3 L'opérateur « crochets » : `[]`

Soit `x` un pointeur sur un type quelconque et `k` une variable de type `int`. Par définition de l'opérateur crochets, `x[k]` est simplement une autre notation, strictement équivalente, pour la variable `*(x+k)` [34]. En particulier, `x[0]` et `*x` sont deux notations pour la même variable.

Remarques

1. Comme l'opérateur `*`, l'opérateur `[]` s'applique aussi aux adresses.
2. Pour le lecteur connaissant les tableaux standards du C, qui ne sont pas étudiés ni utilisés dans ce cours : malgré la similitude de la notation, le `x[k]` défini ici n'est pas un élément de tableau standard du C.

4 Application des pointeurs : transmission à une fonction par adresse

Considérons par exemple la fonction suivante et son appel :

```
int f(int y) {
  ...
}
int main() {
  int x;
  ...
```

32. `k` pourrait être négatif.
33. Sauf, évidemment dans le cas particulier `k=0` où il s'agit de l'emplacement de la variable ordinaire `x`.
34. Il se trouve qu'on peut aussi utiliser la notation `k[x]` au lieu de `x[k]` pour désigner `*(x+k)`. Le compilateur comprend les deux notations. Nous n'utiliserons pas cette possibilité.

```
  ... f(x) ...
  ...
}
```

On a vu au chapitre **Fonctions** que l'appel `f(x)` transmet à `y` la valeur de `x` et que la fonction s'exécute simplement comme si on avait écrit `y = x` à son début. Il y a une façon plus générale de transmettre des arguments à la fonction, à l'aide des adresses et pointeurs. Tout se passe alors comme si on transmettait non seulement des valeurs mais des variables. Jusqu'ici l'argument muet de `f`, `y`, est une variable ordinaire. Il est possible de la remplacer par un pointeur. On a alors :

```
int f(int* y) {
  ...
}
int main() {
  int x;
  ...
  ... f(&x) ...
  ...
}
```

L'appel se fait maintenant en passant l'adresse de `x` puisque `y` est un pointeur. La fonction s'exécute comme si on avait écrit `y = &x` à son début. Par ce procédé `x` et `*y` désignent le même emplacement de mémoire et sont donc deux noms différents pour une même variable, cette variable s'appelant `x` dans le programme principal et `*y` dans la fonction et ceci tout le temps de l'exécution de la fonction. Toute modification de `*y` dans la fonction est automatiquement répercutée sur `x`. On voit qu'il y a là une autre possibilité de retourner des valeurs au programme appelant, par l'intermédiaire des arguments et non plus seulement par le `return` de la fonction. Ce type de transmission est nommé « transmission par adresse » .

Exemple 1

C'est l'exemple déjà vu au chapitre **Fonctions** :

```
#include<iostream>
using namespace std;
int f(int y) {
  y = y+1;
  return y;
}
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
2
1
```

C'est la transmission par valeur, dans le `main`, `x` n'est pas modifié par l'appel de la fonction.

Exemple 2

On remplace l'argument de la fonction `f` par un pointeur :

```
#include<iostream>
using namespace std;
int f(int* y) {
  *y = *y+1;
  return *y;
}
```

```
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(&x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
2
2
```

C'est la transmission par adresse. Durant l'exécution de la fonction `f`, le `x` du `main` est la même variable que le `*y` de la fonction donc il est aussi augmenté de 1 après l'exécution de la fonction.

Remarque marginale mais troublante

Si on écrit le programme précédent en faisant simplement afficher les trois quantités par le même `cout`, en ne changeant rien d'autre :

```
#include<iostream>
using namespace std;
int f(int *y) {
  *y = *y+1;
  return *y;
}
int main() {
  int x;
  x = 1;
  cout << x << f(&x) << x << endl;
  return 0;
}
```

on obtient :

```
2 2 1
```

parce que le calcul des quantités à afficher n'est pas fait dans l'ordre de gauche à droite comme on s'y attendrait.

Exemple 3

On met deux pointeurs au lieu d'un en argument de la fonction `f` :

```
#include<iostream>
using namespace std;
int f(int* y, int* z) {
  *y = *y+1;
  *z = *z+1;
  return *y;
}
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(&x,&x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
3
3
```

Puisqu'on appelle la fonction avec deux fois l'argument effectif `&x` le `x` du `main()` et les `*y` et `*z` de la fonction ne sont qu'une seule et même variable dont la valeur est deux fois augmentée de 1, donc le résultat doit bien être 3.

Exemple d'utilisation du retour de valeur par argument :

On veut résoudre l'équation du second degré :

$$ax^2 + bx + c = 0$$

pour n'importe quelles valeurs réelles des trois paramètres a, b, c. On écrit une fonction à laquelle on fournit ces trois paramètres en argument et ayant la valeur :

0 s'il n'y a pas de racine réelle
1 s'il y a une seule racine réelle (simple ou double)
2 s'il y a deux racines réelles distinctes
3 dans le cas $a = b = c = 0$ (tout x est solution)

La fonction retourne de plus par ses arguments les valeurs des racines quand il y en a.

```
#include<iostream>
#include<math.h>
using namespace std;
int r2(double a, double b, double c, double* xp, double* xs) {
  double delta, rd;
  if (a == 0.)
    if (b == 0.)
      if (c == 0.)
        return 3;
      else
        return 0;
    else {
      *xp = -c/b;
      *xs = *xp;
      return 1;
    }
  else {
    delta = b*b - 4.*a*c;
    if (delta < 0.)
      return 0;
    else if (delta == 0.) {
      *xp = -b/2./a;
      *xs = *xp;
      return 1;
    }
    else {
      rd = sqrt(delta);
      *xp = (-b-rd)/2./a;
      *xs = (-b+rd)/2./a;
      return 2;
    }
  }
}
int main() {
  double x1, x2;
  int ind;
  ind = r2(1., -9., 14., &x1, &x2);
  if (ind == 0) cout << "Pas de racines" << endl;
  if (ind == 1) cout << "Une racine : " << x1 << endl;
  if (ind == 2) cout << "Deux racines : " << x1 << " " << x2 << endl;
  if (ind == 3) cout << "Infinité de racines (pas d'équation)" << endl;
```

```
  return 0;
}
```

5 Pointeur de pointeur

5.1 Définition

Comme une variable ordinaire, un pointeur a une adresse et on peut définir des variables qui contiennent l'adresse d'un pointeur : ce sont des pointeurs de pointeur. Ils sont, par exemple, déclarés de la façon suivante :

```
double** p
```

`p` est un pointeur de pointeur pouvant contenir l'adresse d'un pointeur sur un `double`. Supposons que :

```
double a;
double* b;
double** c;
b = &a;
c = &b;
```

alors :

`a` et `*b` sont deux notations pour la même variable ordinaire (déjà vu)
`b` et `*c` sont deux notations pour le même pointeur
`**c` [35] et `a` sont deux notations pour la même variable ordinaire

On peut ainsi écrire par exemple :

```
#include<iostream>
using namespace std;
int main() {
  double a, *b, **c;
  a = 7.; b = &a; c = &b;
  cout << a << " " << *b << " " << **c << endl;
  return 0;
}
```

qui donne :

```
7 7 7
```

5.2 L'opérateur [] appliqué à un pointeur de pointeur

Définissons :

```
double x, *p, **pp;
p = &x; pp = &p;
```

35. Qui représente `*(*c)`

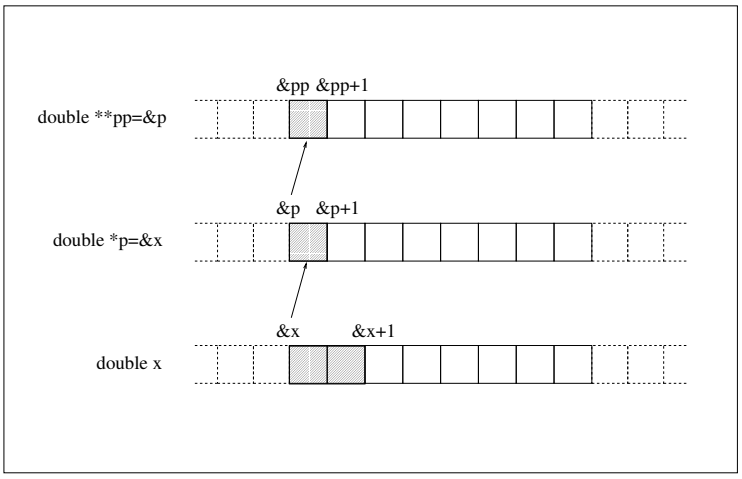


FIGURE 9 – Chaque carré symbolise quatre octets. On suppose ici un système d'exploitation à 32 bits, où les pointeurs ont une taille de 4 octets. Dans un système d'exploitation à 64 bits les pointeurs ont une taille de 8 octets.

On a :

`pp` est un pointeur de pointeur sur un `double`
`pp[i]` et `*(pp+i)` représentent la même variable (qui est un pointeur sur un `double`)
`pp[i][j]` [36] et `*(*(pp+i)+j)` représentent la même variable (qui est une variable `double` ordinaire).

Il faut rappeler que `pp[i]`, `*(pp+i)`, `pp[i][j]` et `*(*(pp+i)+j)`, déclarées comme elles le sont, sont des variables « illégales » , sauf si `i=0` et `j=0`, mais que nous avons choisi de ne pas nous préoccuper de cette question pour l'instant. Dans le cas particulier `i=0` et `j=0`, `x`, `*p`, `p[0]`, `**pp`, `*pp[0]` et `pp[0][0]` représentent la même variable « légale » (qui est une variable `double` ordinaire).

Exemple

```
#include<iostream>
using namespace std;
int main() {
  double x, *p, **pp;
  p = &x; pp = &p;
  x = 5.;
  cout << x << " " << *p << " " << p[0] << " " << **pp << " " << *pp[0]
       << " " << pp[0][0] << endl;
  return 0;
}
```

donne :

```
5 5 5 5 5 5
```

6 Pointeur sur une fonction

6.1 Déclaration d'un pointeur sur une fonction

Soit, par exemple, la fonction `f` de prototype :

```
double f(double);
```

`f` a une adresse et cette adresse peut être attribuée à un pointeur `p` déclaré avec le type correspondant au prototype de `f`, c'est à dire ici :

```
double (*p)(double);
```

36. Qui est équivalent à `(pp[i])[j]`

`p` ne peut pointer que sur des fonctions de type `double` ayant un seul argument de type `double`. Pour une fonction de prototype :

```
double g(double,int);
```

il faut un pointeur déclaré par :

```
double (*q)(double,int);
```

Ceci permet d'utiliser successivement un même nom pour des fonctions différentes, ou si l'on préfère, de définir une « fonction variable » . Par exemple :

```
double f(double x) {
  ...
}
double g(double y) {
  ...
}
int main() {
  double a;
  ...
  double (*p)(double);
  ...
  p = f; /* à partir d'ici (*p)(a), ou plus simplement p(a), est équivalent à f(a) */
  ...
  p = g; /* à partir d'ici (*p)(a), ou plus simplement p(a), est équivalent à g(a) */
  ...
}
```

On remarque sur cet exemple que l'adresse d'une fonction est donnée par `f` et non `&f`.

6.2 Fonction en argument d'une fonction

Reprenons la fonction `integ` qui calcule l'intégrale d'une fonction par la méthode des rectangles, écrite au chapitre **Fonctions**. Elle a un défaut important : elle ne peut, durant une même exécution du programme, intégrer qu'une seule fonction, celle définie par ailleurs avec le nom `f`, puisque ce nom figure explicitement dans `integ`. Si l'on veut intégrer une autre fonction il faut modifier le programme et recompiler. Il serait beaucoup plus général que la fonction `f` puisse varier d'un appel à l'autre de `integ`. Ceci peut être obtenu en mettant `f` en argument à l'aide d'un pointeur. Il suffit d'ajouter dans `integ` un argument muet `double (*f)(double)` qui est un pointeur sur une fonction. Dans `integ`, la fonction à intégrer sera alors désignée par `f` ou `(*f)` [37].
Ce qui donne :

```
double integ(double a, double b, double (*f)(double), int n) {
  double h, s;
  int i;
  h = (b-a)/n;
  s = 0.;
  for (i = 1; i <= n; i++)
    s = s + f(a+(i-0.5)*h));
  return h*s;
}
```

L'utilisation de `integ` peut alors se faire selon l'exemple suivant :

```
#include<iostream>
#include<math.h>
using namespace std;
double f1(double x) {
  return 1./(1.+x*x);
}
double f2(double x) {
  return x*(x-1.);
```

37. Les deux sont possibles.

```
}
double integ(double a, double b, double (*f)(double), int n) {
  double h, s; int i;
  h = (b-a)/n; s = 0.;
  for(i = 1; i <= n; i++)
    s += f(a+(i-0.5)*h);
  return h*s;
}
int main() {
  double a1, b1, a2, b2;
  int np1, np2;
  a1 = 1.; b1 = 2.; np1 = 1000;
  a2 = 0.; b2 = 1.; np2 = 1000;
  cout << integ(a1,b1,f1,np1) << endl;;
  cout << integ(a2,b2,f2,np2) << endl;
  return 0;
}
```

Remarques

1. Dans les arguments muets de l'en-tête d'une fonction il est possible d'enlever l'étoile et les parenthèses dans la déclaration d'un pointeur sur une fonction : `double f(double)` au lieu de `double (*f)(double)` dans cet exemple-ci. Mais cela n'est pas possible si le pointeur sur une fonction est déclaré à l'intérieur d'une fonction. Il n'est donc pas possible de remplacer `double (*p)(double);` par `double p(double);` dans l'exemple de la section précédente. C'est logique : cette dernière notation est la façon de déclarer le prototype d'une fonction.
2. Attention : pour utiliser une fonction `g` dans une fonction `f` il n'est nullement nécessaire de transmettre `g` en argument de `f`. Ce n'est que dans le cas où une fonction `g` utilisée par `f` doit pouvoir changer d'un appel à l'autre de `f` qu'il faut mettre `g` en argument de `f`.
3. Ne pas confondre `f(g)` et `f(g(x))`. Dans le premier cas c'est la fonction `g` qui est transmise en argument à la fonction `f`, tandis que dans le second, c'est la valeur de `g` au point `x` qui est transmise à `f`.
4. Dans le chapitre **Chaînes de caractères** (hors programme) il est décrit comment on peut, de plus, fournir la fonction à intégrer comme la valeur d'une variable lue par un `cin`.

9. Tableaux dynamiques

1 Allocation dynamique de mémoire

1.1 Préliminaire : l'opérateur `sizeof`

Cet opérateur fournit la dimension d'un objet en octets :

`i` étant un `int` : `sizeof(i)` vaut 4
`x` étant un `double` : `sizeof(x)` vaut 8
`c` étant un `char` : `sizeof(c)` vaut 1
`pi` étant un pointeur sur un `int` : `sizeof(pi)` vaut 8
`px` étant un pointeur sur un `double` : `sizeof(px)` vaut 8
`pc` étant un pointeur sur `char` : `sizeof(pc)` vaut 8

On peut aussi avoir directement la taille du type `int`, `double` ou `char` en écrivant :

`sizeof(int)` qui vaut 4
`sizeof(double)` qui vaut 8
`sizeof(char)` qui vaut 1
`sizeof(int*)` qui vaut 8
`sizeof(double*)` qui vaut 8
`sizeof(char*)` qui vaut 8

À noter que ces valeurs peuvent changer d'un ordinateur à un autre et d'un système d'exploitation à un autre. C'est pour cette raison qu'il faut toujours utiliser `sizeof`.

1.2 Allocation de `n` octets

L'allocation dynamique de mémoire permet à l'utilisateur de réserver un emplacement mémoire de `n` octets, par l'intermédiaire de l'adresse du premier de ces octets. La fonction :

```
malloc(n);
```
[38]

réserve `n`[39] octets et sa valeur est l'adresse du premier d'entre eux.
Il est très important de noter que `n` peut être une variable et pas seulement une constante, ce qui justifie l'adjectif « dynamique » .
Le nombre d'octets occupés par un `double` est `sizeof(double)`. Donc, si on veut réserver la place pour `n double` on écrit : `malloc(n*sizeof(double))`.

2 Tableau dynamique à un indice

2.1 Allocation d'un tableau dynamique à un indice

Réservons par exemple la place pour `n double` comme vu à la section précédente :

```
malloc(n*sizeof(double));
```

L'adresse du premier des octets réservés retournée par `malloc` n'a pas de type[40]. Une augmentation de 1 de cette adresse se traduit simplement par une augmentation de 1 du numéro d'octet. Si, par contre, on la convertit en adresse d'un type précis, cette augmentation est égale à la taille de ce type. Convertissons l'adresse `malloc(n*sizeof(double))` en adresse d'un `double` par :

```
(double*)malloc(n*sizeof(double));
```

et alors la valeur de `(double*)malloc(n)+1` sera celle de `(double*)malloc(n)` augmentée de 8 (taille d'un `double`). Plaçons enfin l'adresse fournie par `malloc` convertie en l'adresse d'un `double` dans un pointeur sur un `double` :

```
int n;
double* x;
...
n = 7;
```

38. La fonction `malloc` nécessite la directive `#include<stdlib.h>`
39. En principe `n` doit être de type `size_t` et non de type `int`. On n'en tient pas compte ici.
40. La fonction `malloc` est dite de type « générique » .

```
...
x = (double*)malloc(n*sizeof(double));
```

La variable `*x`, qui peut aussi être notée `x[0]`, est alors une variable de type `double` qui occupe les 8 premiers octets réservés par l'instruction `malloc`. Ce qui est intéressant c'est qu'à la suite de cette variable, il reste `n-1` emplacements mémoire disponibles, d'adresses consécutives, pouvant accueillir chacun une variable de type `double`. Puisque les adresses sont consécutives ces `n-1` variables sont tout simplement `*(x+1)`, `*(x+2)`, ..., `*(x+n-1)`, que, pour la commodité, on notera plutôt à l'aide de la notation `[]` : `x[1]`, `x[2]`, ..., `x[n-1]`.
Par ce procédé l'utilisateur a à sa disposition des variables indicées que l'on appellera tableaux dynamiques pour les distinguer des tableaux standards du C qui ne sont pas étudiés ni utilisés dans ce cours : en effet, excepté pour les cas très simples, l'utilisation des tableaux dynamiques est plus commode, logique et générale que celle des tableaux standards [41].
Sur le plan pratique il est important de retenir que les `n` éléments d'un tableau dynamique sont indicés de `0` à `n-1` et non de `1` à `n`.

En résumé, la forme condensée suivante de ce qui précède :

```
int n = 5;
double* x = (double*)malloc(n*sizeof(double));
```

crée un tableau dynamique de type `double` dont les éléments sont désignés par `x[0]`, `x[1]`, ..., `x[4]`.
La méthode se transpose de façon évidente à des tableaux d'entiers ou de caractères.

Pour libérer l'espace mémoire réservé quand on n'en a plus l'usage, on utilise simplement l'instruction :

```
free(x);
```

Si on ne le fait pas, cet espace mémoire reste inutilisable en pure perte jusqu'à la fin de l'exécution du programme ce qui ne peut que le ralentir et même, dans le cas d'allocations réitérées, conduire à une saturation complète de la mémoire.

2.2 Exemple d'utilisation de tableaux dynamiques à un indice

Les deux programmes suivants calculent le produit scalaire de deux vecteurs, le premier sans utiliser de tableau :

```
#include<iostream>
using namespace std;
int main() {
  double ux, uy, uz, vx, vy, vz, s;
  ...
  s = ux*vx + uy*vy + uz*vz;
  ...
  return 0;
}
```

le second en utilisant un tableau :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int d = 3, i;
  double *u = (double*)malloc(d*sizeof(double)), *v = (double*)malloc(d*sizeof(double)), s;
  ...
  for(s = 0., i = 0; i < d; i++)
    s = s + u[i]*v[i];
  ...
  free(u); free(v);
  return 0;
}
```

41. Une comparaison des deux types de tableaux se trouve aux chapitres (hors programme) **Tableaux standards** et **Rapports entre tableaux standards, adresses et pointeurs**.

Le second programme paraît plus compliqué mais il est plus général : si on veut calculer dans un espace à plus de trois dimensions, il faut, dans le premier cas, créer de plus en plus de noms variables et écrire explicitement ces noms dans les opérations, ce qui peut devenir très lourd voire irréalisable alors qu'il suffit, dans le second cas, de changer la valeur de `d`.

2.3 Dépassement de la taille déclarée

Considérons le programme suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 4;
  double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
  r = 10.;
  for (i = 0; i < n; i++) {
    x[i] = i; y[i] = r+i;
  }
  for (i = 0; i < n; i++)
    cout << x[i] << " " << y[i] << endl;
  free(x); free(y);
  return 0;
}
```

dont le résultat est :

```
0 10
1 11
2 12
3 13
```

Si on déclare `x` et `y` comme de simples variables et non des pointeurs :

```
double r, x, y;
```

au lieu de :

```
double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
```

on a le diagnostic suivant à la compilation : *erreur : invalid types 'double[int]' for array subscript*
Mais si on excède la valeur déclarée de la dimension, par exemple :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 4, p = 6;
  double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
  r = 10.;
  for (i = 0; i < p; i++) {
    x[i] = i; y[i] = r+i;  // cette boucle dépasse la dimension déclarée pour x (p>n)
  }
  for (i = 0; i < p; i++)
    cout << x[i] << " " << y[i] << endl; // idem
  free(x); free(y);
  return 0;
}
```

soit il y a un diagnostic à l'exécution tel que *Erreur de segmentation*, soit il n'y a aucun diagnostic et c'est très dangereux car un résultat est fourni mais il peut être complètement faux sans que l'utilisateur s'en aperçoive.
Il faut donc, à l'écriture des programmes, veiller de façon extrêmement vigilante à ne pas dépasser les dimensions déclarées pour les tableaux dynamiques.

2.4 Initialisation et affichage des tableaux

Les éléments des tableaux dynamiques ne sont pas initialisés à une valeur donnée lors de l'allocation de mémoire. Ils contiennent simplement la valeur correspondant à l'état, complètement indéterminé pour l'utilisateur, dans lequel se trouvaient les registres de mémoire avant l'allocation. Dans le cas où c'est nécessaire, l'utilisateur doit donc initialiser le tableau.

2.4.1 Exemples d'allocation, d'initialisation et d'affichage d'un tableau à un indice (vecteur)

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 5;
  double* x = (double*)malloc(n*sizeof(double));
  cout << "Initialisation d'un vecteur à 0 :" << endl;
  for (i = 0; i < n; i++)
    x[i] = 0.;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  cout << "Initialisation d'un vecteur à une suite de carrés d'entiers :" << endl;
  for (i = 0; i < n; i++)
    x[i] = i*i;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  free(x);
  return 0;
}
```

ce qui donne :

Initialisation d'un vecteur à 0 :
0 0 0 0 0
Initialisation d'un vecteur à une suite de carrés d'entiers :
0 1 4 9 16

2.4.2 Cas où les valeurs d'initialisation sont données explicitement

Quand les valeurs d'initialisation ne peuvent être exprimées par une formule programmable mais sont fournies explicitement (par exemple ce sont des valeurs expérimentales provenant d'une série de mesures) on a le choix entre trois méthodes :

1. Dans le cas où leur nombre se réduit à quelques unités, écrire un à un les éléments de tableau :

   ```
   x[0] = 1.32; x[1] = -0.37; x[2] = 6.78;
   ```

2. Dans le cas où leur nombre ne dépasse pas quelques dizaines, utiliser une fonction d'initialisation sur le modèle suivant :

   ```
   #include<iostream>
   #include<stdlib.h>
   #include<stdarg.h>
   using namespace std;
   void ini(double* x, int ni, ...) {
     int i;
     va_list ap;
     va_start(ap,ni);
     for (i = 0; i < ni; i++)
       x[i] = va_arg(ap,double);
   ```

```
  va_end(ap);
}
/*************************************************************************/
int main() {
  int i, nn = 3;
  double* t = (double*)malloc(nn*sizeof(double));
  ini(t, nn, 2.3, -4.1, 5.7);
  for (i = 0; i < nn; i++)
    cout << t[i] << " ";
  cout << endl;
  free(t);
  return 0;
}
```

La fonction nommée ici `ini` reçoit en argument :

le nom du tableau dynamique à initialiser
la dimension de ce tableau
la liste des valeurs d'initialisation

C'est une fonction à nombre variable d'arguments [42], dont le principe a été exposé au chapitre **Fonctions**. De telles fonctions sont déjà écrites et disponibles dans la bibliothèque des fonctions du Magistère, leur utilisation est décrite à l'annexe située en fin de ce chapitre.

Remarque

Dans une telle fonction à nombre variable d'arguments il est très important de respecter strictement le type des arguments. Si les arguments sont du type double et on veut passer la valeur 2 à la fonction, il faut l'écrire comme `2.` avec un point. Dans ce cas il n'y a pas de conversion automatique de type, comme pour les fonctions avec un nombre fixe d'arguments, et si l'on ne respecte pas les types le résultat sera complètement aberrant.

3. Dans les autres cas écrire préalablement les valeurs d'initialisation dans un fichier de données puis les faire lire et placer dans les éléments de tableau par le programme.

2.5 Tableau dynamique à un indice en argument d'une fonction

Pour transmettre un tableau dynamique à une fonction il suffit de transmettre :
— l'adresse du premier élément du tableau dans un pointeur argument muet de la fonction
— le nombre d'éléments du tableau.

Prenons l'exemple d'une fonction calculant la norme d'un vecteur à trois composantes. Cette fonction et son appel par le `main` peuvent s'écrire :

```
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
//-------------------------------------
double norme(double* x, int p) {
  int i; double s;
  for (s = 0., i = 0; i < p; i++)
    s = s + x[i]*x[i];
  return sqrt(s);
}
//-------------------------------------
int main() {
  int i, n1 = 3, n2 = 5;
  double* v1 = (double*)malloc(n1*sizeof(double));
  for (i = 0; i < n1; i++)
    v1[i] = i;
```

42. Rappel : il n'est pas demandé aux étudiants de maîtriser les fonctions de ce type, mais la connaissance de leur existence et de leur utilisation peut être utile en pratique, comme ici par exemple.

```
  double* v2 = (double*)malloc(n2*sizeof(double));
  for (i = 0; i < n2; i++)
    v2[i] = i+3;
  cout << norme(v1,n1) << endl;
  cout << norme(v2,n2) << endl;
  free(v1); free(v2);
  return 0;
}
```

Lors de l'appel `norme(v1,n1)` l'adresse contenue dans le pointeur `v1`, qui est celle de l'élément `v1[0]`, est mise dans le pointeur `x`, comme si on avait écrit `x = v1`[43]. Le pointeur `x` contenant la même adresse que le pointeur `v1`, il donne accès, dans la fonction, par l'intermédiaire de l'opérateur `[]`, aux mêmes emplacements de mémoire que le pointeur `v1` : `x[0]` et `v1[0]` correspondent au même emplacement de mémoire et contiennent donc la même valeur, de même pour `x[1]` et `v1[1]`, etc. Durant l'exécution de la fonction, `x` et `v1` sont deux noms différents pour le même tableau. Dans la transmission à la fonction il n'y a pas duplication des valeurs des éléments de `v1` dans `x` mais simplement passage de l'adresse du premier élément de `v1`.
Ceci a pour conséquence que toute modification des éléments de `x` dans la fonction entraîne automatiquement celle des éléments de `v1` : s'il y a modification de `x`, le tableau `v1` ne contient plus, après l'appel de la fonction, les mêmes valeurs que celles qu'il contenait avant, l'argument effectif est modifié. Ce mécanisme peut entraîner des effets indésirables si l'on n'y prend pas garde, mais peut inversement être utilisé de façon avantageuse pour transmettre des valeurs dans le sens fonction appelée vers fonction appelante et non plus seulement dans le sens fonction appelante vers fonction appelée. Par exemple, la fonction suivante reçoit en entrée un vecteur quelconque dans un tableau et retourne le vecteur unitaire parallèle et de même sens dans le même tableau :

```
//vecteur_unitaire_un_vecteur.cpp
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
int f(double* v, int n) {
  double c, q;
  int i;
  for (c = 0., i = 0; i < n; i++)
    c = c + v[i]*v[i];
  q = sqrt(c);
  if (q == 0.)
    return 0;
  for (i = 0; i < n; i++)
    v[i] = v[i]/q;
  return 1;
}
int main() {
  int i, n = 2;
  double* x = (double*)malloc(n*sizeof(double));
  x[0] = 1.; x[1] = 2.;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  f(x,n);
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  free(x);
  return 0;
}
```

Si on veut que le vecteur d'entrée ne soit pas modifié il faut utiliser deux pointeurs :

43. Cela se passe comme pour des variables ordinaires : la valeur contenue dans la variable argument effectif est placée dans la variable argument muet.

```
//vecteur_unitaire_deux_vecteurs.cpp
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
int f(const double* v, double* u, int n) // on met const pour être sûr que
{                                        // v ne sera pas modifié
  double c, q;
  int i;
  for(c = 0., i = 0; i < n; i++)
    c = c + v[i]*v[i];
  q = sqrt(c);
  if(q == 0.) {
    for(i = 0; i < n; i++)
      u[i] = 0;
    return 0;
  }
  for(i = 0; i < n; i++)
    u[i] = v[i]/q;
  return 1;
}
int main() {
  int i, n = 2;
  double* x = (double*)malloc(n*sizeof(double));
  double* xu = (double*)malloc(n*sizeof(double));
  x[0] = 1.; x[1] = 2.;
  f(x,xu,n);
  for(i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  for(i = 0; i < n; i++)
    cout << xu[i] << " ";
  cout << endl;
  free(x); free(xu);
  return 0;
}
```

3 N'employer un tableau que lorsque c'est réellement nécessaire

On ne doit employer un tableau que s'il faut conserver des valeurs pour les réutiliser en un autre point du programme. Sinon on mobilise des mémoires pour rien et, de plus, on subit les contraintes liées à la déclaration.

Exemple

On veut calculer les n premiers termes de la suite : $u_n = au_{n-1} + bu_{n-2}$, connaissant u_0 et u_1.

1) Avec un tableau :

```
/* Calcul de la suite u(n)=a*u(n-1)+b*u(n-2) en utilisant inutilement un tableau */
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, nn = 50;
  double a = 5.6, b = -3.4, *u = (double*)malloc(nn*sizeof(double));
  u[0] = -4.2; u[1] = 7.1;
  for(i = 2; i < nn; i++)
    u[i] = a*u[i-1] + b*u[i-2];
  for(i = 0; i < nn; i++)
    cout << "u(" << i << ")=" << u[i] << endl;
```

```
  free(u);
  return 0;
}
```

Cela ne sert à rien de stocker les valeurs de u_n dans un tableau puisqu'elles sont affichées au fur et à mesure à l'écran et qu'on ne s'en sert pas ensuite dans le programme.

2) Sans tableau :

```
/* Calcul de la suite u(n)=a*u(n-1)+b*u(n-2) sans utiliser de tableau */
#include<iostream>
using namespace std;
int main() {
  int i, nn = 50;
  double a = 5.6, b = -3.4, u, u0 = -4.2, u1 = 7.1;
  cout << "u(0)=" << u0 << endl;
  cout << "u(1)=" << u1 << endl;
  for(i = 2; i < nn; i++) {
    u = a*u1 + b*u0;
    cout << "u(" << i << ")=" << u << endl;
    u0 = u1; u1 = u;
  }
  return 0;
}
```

La seconde méthode est plus simple.
Mais si on a besoin des u_n pour une tâche qui ne peut être effectuée au vol (par exemple les faire afficher par valeur croissante) il faut utiliser la première méthode.
On dit que dans le premier cas on a un accès direct aux u_n alors que dans le second on a seulement un accès séquentiel.

Autre exemple

On veut calculer la moyenne et l'écart-type d'un ensemble de valeurs entrées au clavier par l'utilisateur :

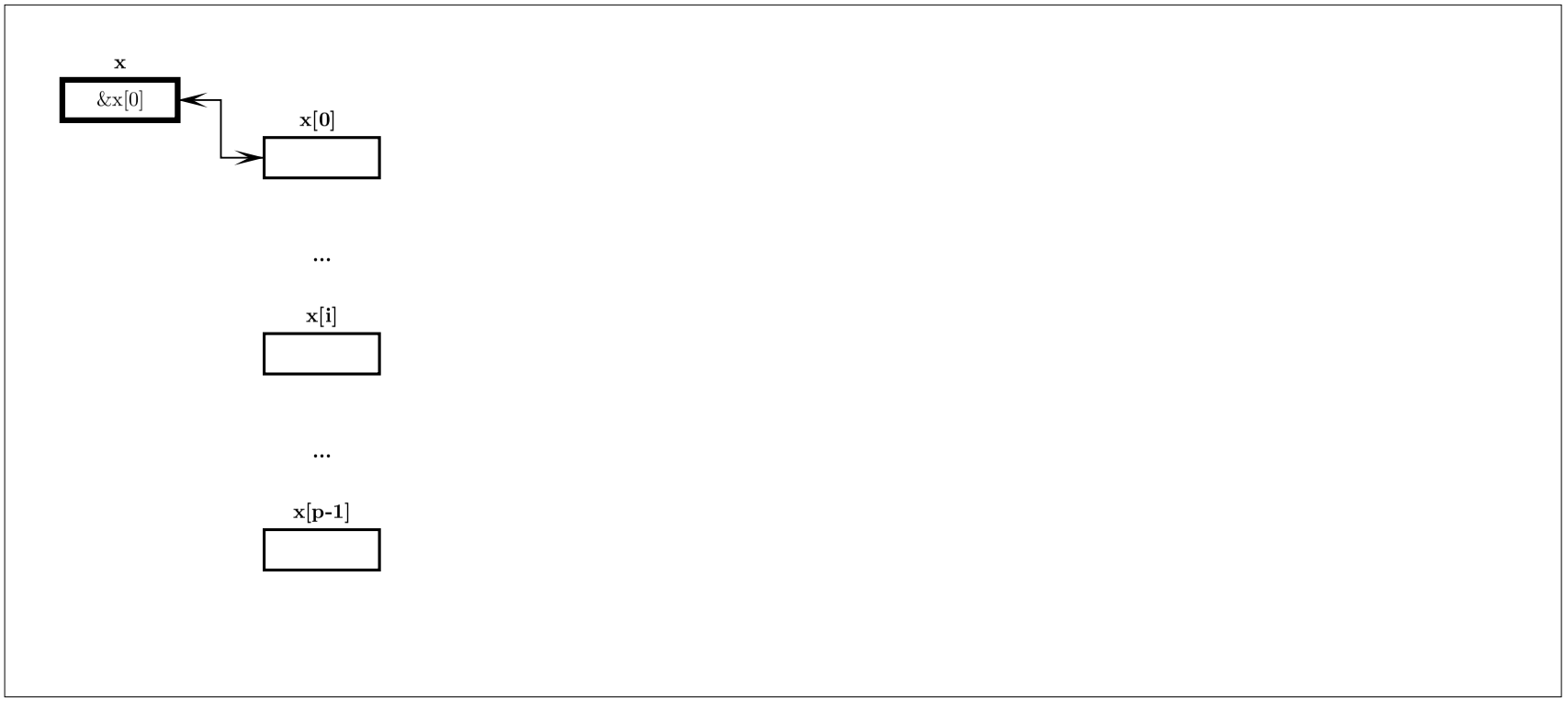
```
#include<iostream>
#include <math.h>
using namespace std;
int main() {
  int i, n;
  double s, s2, x, moy;
  cout << "Nombre de valeurs ? "; cin >> n;
  for(s = 0, s2 = 0, i = 1; i <= n; i++) {
    cin >> x;
    s += x; s2 += x*x;
  }
  moy = s/n;
  cout << "moyenne=" << moy << "  écart-type=" << sqrt(s2/n-moy*moy) << endl;
  return 0;
}
```

On voit qu'il est inutile de stocker les valeurs dans un tableau.

4 Tableaux dynamiques à plus d'un indice

4.1 Allocation d'un tableau dynamique à plus d'un indice

Les deux figures suivantes illustrent les explications de cette section :

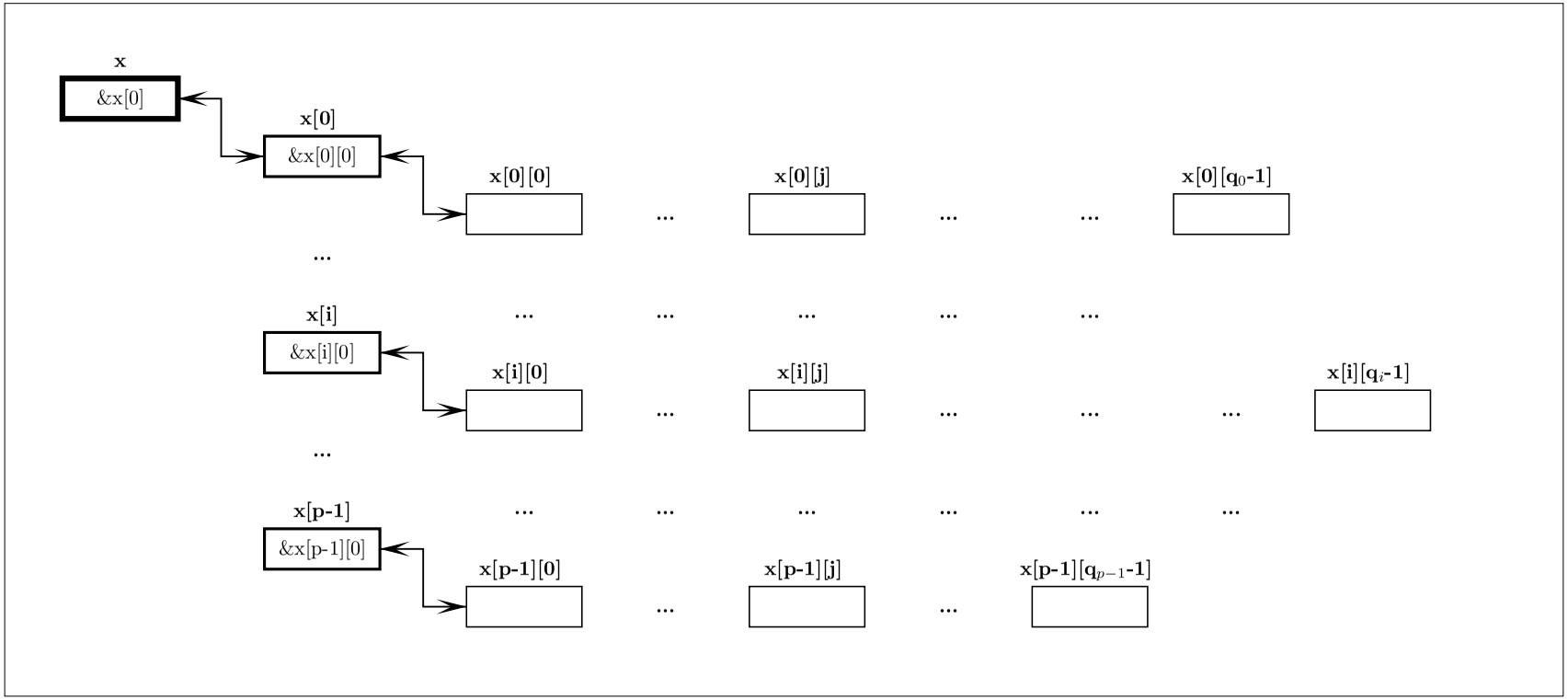


Les instructions ci-contre produisent l'allocation de mémoire schématisée ci-dessus.

```
double **x ;
int p ;
p=... ;
x=(double **)malloc(p*sizeof(double *)) ;
```

A l'issue de cette première étape on a créé `p` pointeurs de `double` : `x[0] ... x[p-1]`, dans lesquels on n'a pour l'instant mis aucune adresse.

Première étape de l'allocation de mémoire pour un tableau-pointeur à deux indices



Les instructions ci-contre complètent celles de la première étape pour obtenir l'allocation de mémoire d'un tableau-pointeur à deux indices schématisée ci-dessus.
On peut choisir d'avoir ou non des lignes toutes de la même longueur. Pour avoir des lignes toutes de longueur q il suffit faire $q_0 = \cdots = q_i = \cdots = q_{p-1} = q$, on obtient alors un tableau-pointeur rectangulaire.

```
int q0, ... qi, ... qp-1 ;
q0=... ; ... qi=... ; ... qp-1=... ;
x[0]=(double *)malloc(q0*sizeof(double)) ;
...
x[i]=(double *)malloc(qi*sizeof(double)) ;
...
x[p-1]=(double *)malloc(qp-1*sizeof(double)) ;
```

A l'issue de cette seconde étape on a mis des adresses de `double` dans chacun des pointeurs de `double` : `x[0] ... x[p-1]` et on a créé `q0 + ... + qi + ... + qp-1` variables ordinaires de type `double` auxquelles on n'a pour l'instant attribué aucune valeur.

Seconde étape de l'allocation de mémoire pour un tableau-pointeur à deux indices

Considérons la déclaration :

```
double** x;
```

`x` est un pointeur de pointeur de `double` susceptible de recevoir l'adresse d'un pointeur de `double` `*x` ou `x[0]` est un pointeur de `double` susceptible de recevoir l'adresse d'un `double`.

Ajoutons maintenant à la déclaration précédente :

```
int p;
...
p = ...;
x = (double**)malloc(p * sizeof(double*));
```

cette dernière instruction met dans `x` une adresse de pointeur de `double` à partir de laquelle `p` emplacements de pointeurs de `double` sont réservés.
Ces `p` pointeurs de `double` sont alors désignés par `*x`, `*(x+1)`, ..., `*(x+p-1)` ou `x[0]`, `x[1]`, ..., `x[p-1]` [44]. Chacun de ces `p` pointeurs peut servir à créer un tableau dynamique à un indice en ajoutant par exemple aux instructions précédentes :

```
int q0, q1, ...;
...
q0 = ...; q1 = ...; ...
x[0] = (double*)malloc(q0 * sizeof(double));
x[1] = (double*)malloc(q1 * sizeof(double));
...
```

On peut choisir, comme cas particulier : `q0=q1=q2=...=q` et les instructions précédentes s'écrivent :

```
int q;
...
q = ...;
for(i = 0; i < p; i++)
  x[i] = (double*)malloc(q * sizeof(double));
...
```

`x[i][j]` (avec `i=0 ... p-1` et `j=0 ... q-1`) est alors une variable de type `double` dont l'adresse est `x[i]+j`, l'adresse du pointeur de `double` `x[i]` étant `x+i`.
On a ainsi créé un tableau dynamique à deux indices qui est une pure généralisation du tableau dynamique à un indice et il s'utilise de la même façon.
Pour libérer l'espace mémoire réservé, il faut procéder dans l'ordre inverse comparé aux `malloc` : d'abord faire des `free(x[i])` et puis un `free(x)` :

```
for(i = 0; i < p; i++)
  free(x[i]);
free(x);
```

4.1.1 Exemples d'allocation, d'initialisation et d'affichage d'un tableau à deux indices (matrice)

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, j, n = 5;
  double** x = (double**)malloc(n * sizeof(double*));
  for(i = 0; i < n; i++)
    x[i] = (double*)malloc(n * sizeof(double));
  cout << "Initialisation d'une matrice carrée à 0 :" << endl;
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      x[i][j] = 0.;
  for(i = 0; i < n; i++) {
```

44. A partir d'ici on choisit de n'utiliser que la notation « crochets » .

```
    for(j = 0; j < n; j++)
      cout << x[i][j] << " ";
    cout << endl;
  }
  cout << "Initialisation d'une matrice carrée à l'unité :" << endl;
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      if(i == j)
        x[i][j] = 1.;
      else
        x[i][j] = 0.;
  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++)
      cout << x[i][j] << " ";
    cout << endl;
  }
  for(i = 0; i < n; i++)
    free(x[i]);
  free(x);
  return 0;
}
```

ce qui donne :

Initialisation d'une matrice carrée à 0 :
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Initialisation d'une matrice carrée à l'unité :
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1

L'allocation, l'initialisation et l'affichage pourront être rendus simples et concis par l'utilisation de fonctions.

4.1.2 Cas où les valeurs d'initialisation sont données explicitement

Mêmes méthodes que pour un seul indice. Voir fonctions à l'annexe située à la fin de ce chapitre.

4.2 Tableau dynamique à plus d'un indice en argument d'une fonction

Remarque préliminaire

A l'allocation du tableau dynamique à deux indices `x` faite dans la sous-section précédente ajoutons les instructions :

```
double** y;
y = x;
```

`y` contient alors la même adresse de pointeur de `double` que `x`, `x[i]` et `y[i]` désignent le même emplacement de pointeur de `double` donc le même pointeur de `double`, `x[i][j]` et `y[i][j]` désignent le même emplacement de `double` donc le même `double`, `x` et `y` sont donc deux noms différents pour le même tableau dynamique à deux indices, toute modification des éléments de l'un est mécaniquement appliquée à ceux de l'autre.
On le vérifie sur l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
#include<iomanip>
using namespace std;
int main() {
  double **x, **y;
  int i, j, p, q;
  p = 3; q = 4;
  // On utilise le pointeur de pointeur x pour déclarer un tableau dynamique à deux indices
  x = (double**)malloc(p * sizeof(double*));
  for(i = 0; i < p; i++)
    x[i] = (double*)malloc(q * sizeof(double));
  //-------------------------------------------------------------------------
  // On donne des valeurs quelconques aux éléments du tableau dynamique x
  for(i = 0; i < p; i++)
    for(j = 0; j < q; j++)
      x[i][j] = i*q + j;
  // On fait afficher les valeurs de x
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << x[i][j];
    cout << endl;
  }
  // On met dans y l'adresse qui se trouve dans x
  y = x; // On pourrait aussi bien mettre cette instruction ci-dessus, juste après
         // x = (double**)malloc(p * sizeof(double*));
  // On fait afficher les valeurs de y
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << y[i][j];
    cout << endl;
  }
  //-------------------------------------------------------------------------
  // On donne des valeurs quelconques aux éléments du tableau dynamique y
  for(i = 0; i < p; i++)
    for(j = 0; j < q; j++)
      y[i][j] = i + j*p;
  // On fait afficher les valeurs de y
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << y[i][j];
    cout << endl;
  }
  // Bien noter qu'ici on n'écrit pas x = y; qui serait possible mais inutile
  // On fait afficher les valeurs de x
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << x[i][j];
    cout << endl;
  }
  //-------------------------------------------------------------------------
  // Libérer la mémoire réservée; on pourrait de façon équivalente
  // utiliser free(y[i]) et free(y)
```

```
    for(i = 0; i < p; i++)
      free(x[i]);
    free(x);
    return 0;
  }
```

Considérons maintenant l'exemple de l'appel de la fonction `f` suivante par le `main` :

```
... f(..., double** y, int p, int q, ...) {
  ...
}
//-------------------------------------------------------
int main() {
  double** x;
  int i, p, q;
  ...
  p = ...; q = ...;
  x = (double**)malloc(p*sizeof(double*));
  ...
  for(i = 0; i < p; i++)
    x[i] = (double*)malloc(q*sizeof(double));
  ...
  ... f(..., x, p, q, ...) ...;
  ...
}
```

L'appel de la fonction `f` avec l'argument effectif `x` revient à faire `y = x;`
Le temps de l'exécution de `f` :

- `y` contient la même adresse que `x`
- `y` donne accès aux mêmes emplacements mémoire que `x`, emplacements que l'on peut atteindre par la notation « crochets » `y[i][j]`
- travailler sur les éléments `y[i][j]` est strictement équivalent à travailler sur les éléments `x[i][j]`.

5 Allocation automatisée des tableaux dynamiques

L'emploi des tableaux dynamiques tel que présenté jusqu'ici est assez lourd. On peut le simplifier en utilisant une propriété à priori non évidente de l'allocation dynamique : si elle est faite dans une fonction elle demeure après la fin de l'exécution de cette fonction. Ceci est illustré par l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
double* allo() {
  int i, n = 5;
  double* z = (double*)malloc(n*sizeof(double));
  for(i = 0; i < 5; i++)
    z[i] = i;
  for(i = 0; i < 5; i++)
    cout << z[i] << endl;
  cout << endl;
  return z;
}
int main() {
  int i;
  double* x;
  x = allo();
  for(i = 0; i < 5; i++)
    cout << x[i] << endl;
  free(x);
```

```
  return 0;
}
```

qui donne le résultat :

```
0
1
2
3
4

0
1
2
3
4
```

On vérifie que l'emplacement mémoire réservé dans la fonction `allo` le reste après la fin de l'exécution de cette fonction[45]. En contre-partie, puisque chaque appel de la fonction alloue un nouvel emplacement mémoire, il ne faut pas oublier de désallouer dès que c'est possible par l'instruction `free`, sans quoi la place inutilement réservée peut croître jusqu'à saturer la mémoire.
On peut alors automatiser l'allocation d'un tableau dynamique en utilisant cette propriété comme dans l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
double* allo(int n) {
  double* z = (double*)malloc(n * sizeof(double));
  return z;
}
int main() {
  int n;
  n = ...;
  double* x = allo(n);
  ...
  return 0;
}
```

et l'allocation d'un tableau dynamique se fait donc simplement par l'instruction : [46]

```
double* x = allo(n);
```

Dans `allo` on peut ajouter un test vérifiant que l'allocation s'est faite correctement :

```
double* allo(int n) {
  double* z = (double*)malloc(n * sizeof(double));
  if(z == NULL) {
    cerr << "Mémoire insuffisante" << endl;
    exit -1;
  }
  return z;
}
```

`cerr` est la sortie standard d'erreur, qui permet de séparer les diagnostics d'erreur à l'exécution des sorties normales produites par `cout`.
De même on écrit une fonction pour désallouer :

```
void desallo(double* z) {
  free(z);
}
```

45. Cet exemple n'est pas à lui seul une preuve : l'emplacement mémoire pourrait « par hasard » ne pas avoir été ré-utilisé.
46. Une version alternative de la fonction `allo`, renvoyant l'adresse en argument et non par la valeur de la fonction, serait : `void allo(double** z, int n) {*z = (double*)malloc(n * sizeof(double));}` Remarque bien les `*` supplémentaires ! Dans ce cas l'allocation d'un tableau dynamique se ferait par : `double* x; allo(&x, n);`

et dans le programme appelant il suffit d'écrire :

```
desallo(x)
```

pour libérer l'espace mémoire.

Remarque

Dans ce cas la fonction `desallo` est inutile, il suffit en effet d'écrire directement `free(x)`. Mais, à partir d'un tableau dynamique à deux indices, il devient intéressant d'en disposer.

Des fonctions bâties sur le principe de `allo` et `desallo` pour l'allocation et la désallocation de tableaux dynamiques de type `char`, `int` ou `double` à un, deux ou trois indices et de noms respectifs :

`C_1`, `C_2`, `C_3`, `I_1`, `I_2`, `I_3`, `D_1`, `D_2`, `D_3`
`f_C_1`, `f_C_2`, `f_C_3`, `f_I_1`, `f_I_2`, `f_I_3`, `f_D_1`, `f_D_2`, `f_D_3`

ainsi que des fonctions destinées à l'initialisation des tableaux dynamiques, de noms respectifs :

`ini_C_1`, `ini_C_2`, `ini_C_3`, `ini_I_1`, `ini_I_2`, `ini_I_3`, `ini_D_1`, `ini_D_2`, `ini_D_3`

sont déjà écrites et mises à la disposition des utilisateurs dans la bibliothèque des fonctions du Magistère. Leur mode d'emploi est décrit à l'annexe située à la fin de ce chapitre.

Annexe : fonctions simplifiant l'usage des tableaux dynamiques

On rappelle qu'on appelle dimension du $n^{ième}$ indice d'un tableau dynamique le nombre de valeurs prises par cet indice.
Pour simplifier l'allocation, l'initialisation et la désallocation explicite des tableaux dynamiques, des fonctions ont été écrites et sont disponibles dans la bibliothèque des fonctions du Magistère. Leurs noms sont les suivants :

	Allocation			Initialisation			Désallocation		
Nombre d'indices	Type du tableau			Type du tableau			Type du tableau		
	`char`	`int`	`double`	`char`	`int`	`double`	`char`	`int`	`double`
1	C_1	I_1	D_1	ini_C_1	ini_I_1	ini_D_1	f_C_1	f_I_1	f_D_1
2	C_2	I_2	D_2	ini_C_2	ini_I_2	ini_D_2	f_C_2	f_I_2	f_D_2
3	C_3	I_3	D_3	ini_C_3	ini_I_3	ini_D_3	f_C_3	f_I_3	f_D_3

Leur mode d'emploi est exposé dans la suite de cette annexe.

Allocation des tableaux dynamiques

Les encadrés suivants décrivent l'allocation, l'initialisation explicite éventuelle et la désallocation de tableaux dynamiques de type `double` pour un, deux ou trois indices.
Pour les types `int` ou `char`, il suffit de remplacer `double` par `int` ou `char` et `D` par `I` ou `C`.

Un indice

```
#include<bibli_fonctions.h>
...
int p;
p = ...;
double* x = D_1(p);
ini_D_1(x, p, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_1(x,p);
...
```

Deux indices

```
#include<bibli_fonctions.h>
...
int p, q;
p = ...; q = ...;
double** x = D_2(p,q);
ini_D_2(x, p, q, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_2(x,p,q);
...
```

Pour l'initialisation les valeurs de la séquence `4.5,-6.4,...` sont attribuées aux éléments de tableau dans l'ordre `x[0][0]`, `x[0][1]`, `x[0][2]`, ..., `x[1][0]`, `x[1][1]`, `x[1][2]`, etc., c'est à dire en faisant varier le second, et donc dernier, indice en premier.

Trois indices

```
#include<bibli_fonctions.h>
...
int p, q, r;
p = ...; q = ...; r = ...;
double*** x = D_3(p,q,r);
ini_D_3(x, p, q, r, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_3(x,p,q,r);
...
```

Pour l'initialisation les valeurs de la séquence `4.5,-6.4,...` sont attribuées aux éléments de tableau selon la même règle que pour les tableaux à deux indices, c'est à dire en faisant varier le dernier indice en premier et l'avant dernier en second.

10. Générateur aléatoire uniforme sur [0,1], applications

1 Préliminaire arithmétique

(Peut être sauté en première lecture)

1.1 Division euclidienne

On note $\mathbb{N}$ l'ensemble des entiers naturels. La division euclidienne est l'opération qui, au couple d'entiers $a \in \mathbb{N}$, $b \in \mathbb{N}^*$ fait correspondre le couple d'entiers unique $k \in \mathbb{N}$, $r \in \mathbb{N}$, tel que $a = kb + r$, avec $0 \leq r \leq b - 1$.
a est le dividende, b le diviseur, k le quotient et r le reste.

1.2 Définition d'une suite u_n

On choisit un entier positif q, premier, et un entier p tel que $0 < p < q$.
On définit une suite d'entiers u_n par un terme de départ $u_0 \in [1, q-1]$ (u_0 est appellé le « germe ») et la relation de récurrence : pour $n > 0$, u_n est le reste de la division euclidienne de pu_{n-1} par q. On a donc $pu_{n-1} = kq + u_n$, $k \in \mathbb{N}$.
Cette définition peut aussi s'écrire $u_n = p\,u_{n-1}$ modulo q ou, autre notation, $u_n \equiv p\,u_{n-1}(q)$.
Dans tous les exemples qui suivent on choisit $u_0 = 1$, on verra que cela n'a pas beaucoup d'importance.
La figure ci-dessous donne une représentation graphique d'une telle suite dans le cas $q = 11$, $p = 3$, $u_0 = 1$:

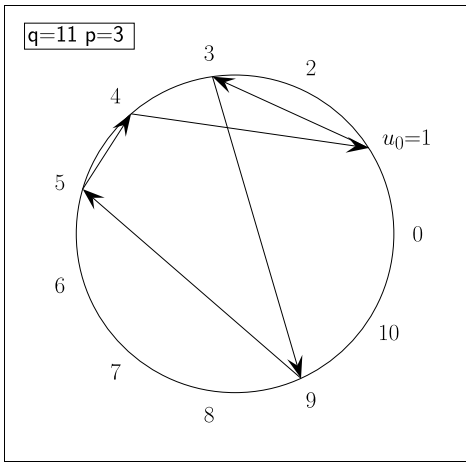


FIGURE 10 –

1.3 Trois propriétés de la suite u_n

1. u_n n'est jamais nul, $\forall n \in \mathbb{N}$.
   A priori les termes de la suite apartiennent à l'intervalle $[0, q-1]$ puisqu'ils sont le reste d'une division par q. Supposons qu'un des termes de la suite, noté u_n, soit nul :

   $$pu_{n-1} = kq \Longrightarrow pu_{n-1} \text{ est divisible par } q$$

   Or si un nombre premier divise un produit il divise l'un des facteurs : q divise donc p ou u_{n-1}. Mais on a choisi p strictement inférieur à q et non nul donc q ne peut le diviser. D'autre part, par définition, u_{n-1} vaut au maximum $q-1$, la seule solution pourqu'il soit divisible par q est donc qu'il soit nul. On a donc $u_n = 0 \Longrightarrow u_{n-1} = 0$, et, par récurrence décroissante, $u_0 = 0$, ce qui est contraire au choix $u_0 \in [1, q-1]$.

   Remarque

   Si q n'était pas premier la suite u_n pourrait être nulle à partir d'un certain rang, comme dans l'exemple suivant : $q = 8$, $p = 6$, $u_0 = 1$, $u_1 = 6$, $u_2 = 4$, $u_3 = 0$, $u_4 = 0$, etc.

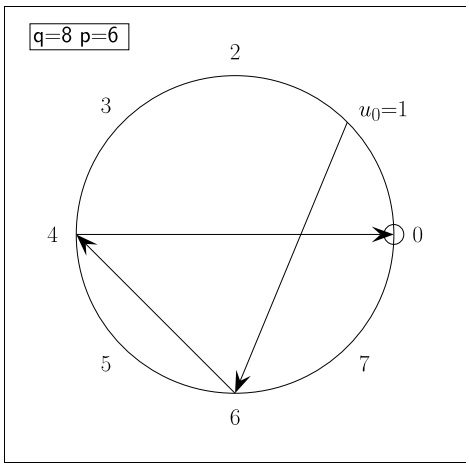


FIGURE 11 –

L'ensemble des valeurs possibles de u_n est donc compris dans l'intervalle $[1, q-1]$.

2. u_n est périodique et la valeur u_0 est la première à être prise une seconde fois.
Puisque u_n ne peut prendre au plus que $q-1$ valeurs différentes et qu'il est déterminé de façon unique par u_{n-1} la suite u_n est nécessairement périodique à partir d'un certain rang, avec une période inférieure ou égale à $q-1$.
D'autre part il est impossible que deux valeurs différentes de u_{n-1}, notées u' et u'', donnent le même u_n. En effet, il existerait alors k' et k'' tels que :

$$pu' = k'q + u_n \text{ et } pu'' = k''q + u_n$$

ce qui implique :

$$p(u'' - u') = (k'' - k')q$$

donc que q divise $p(u'' - u')$ et donc que q divise $u'' - u'$ (puisqu'on a vu qu'il ne peut diviser p). Or :

$$1 \leq u' \leq q-1 \text{ et } 1 \leq u'' \leq q-1 \Longrightarrow |u'' - u'| \leq q-2$$

car le maximum de $|u'' - u'|$ est obtenu pour $u' = 1$ et $u'' = q-1$ (ou l'inverse). La seule possibilité pour que q divise $u'' - u'$ est donc :

$$u'' = u'$$

Soit u_n une valeur prise une seconde fois. D'après ce qui vient d'être démontré précédemment elle a nécessairement le même prédécesseur u_{n-1} que la première fois, donc u_{n-1} a été prise une seconde fois avant u_n. En poursuivant ce raisonnement par récurrence décroissante on aboutit à u_0 qui est donc la première valeur à être prise une seconde fois. La suite u_n n'est donc pas seulement périodique à partir d'un certain rang mais périodique dès le terme de départ u_0. Ce ne serait pas toujours le cas si q n'était pas premier comme le montre l'exemple $q = 20$, $p = 18$, $u_0 = 1$.

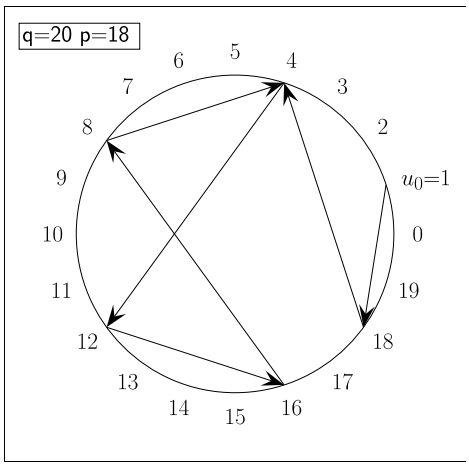


FIGURE 12 –

3. Il existe toujours des valeurs de p telles que u_n prend toutes les valeurs de $[1, q-1]$
   On peut montrer[47] qu'il existe toujours des valeurs particulières de p, appelées « racines primitives » de q, telles que u_n prend toutes les valeurs de l'intervalle $[1, q-1]$, la période ayant alors sa valeur maximum $q-1$.
   La figure suivante montre que le cas $p = 2$ est une racine primitive pour $q = 11$ puisque toutes les valeurs de l'intervalle $[1, 10]$ sont atteintes :

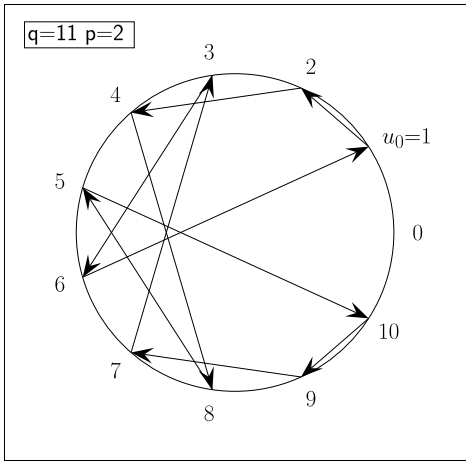


FIGURE 13 –

   Si p est une racine primitive, on voit que changer la valeur de u_0 fait décrire la même séquence cyclique mais à partir d'un point de départ différent[48].

La séquence des valeurs de u_n ayant un aspect désordonné, on considère qu'elle simule une suite d'entiers tirés au hasard avec une probabilité uniforme entre 1 et $q-1$[49].
Pour calculer les racines primitives on procède par essais successifs : pour p et q donnés on calcule les u_n jusqu'à retrouver u_0. Si le nombre de termes différents obtenus est égal à $q-1$, p est une racine primitive.

Exemple

Pour $q = 11$ on calcule la suite des u_n pour toutes les valeurs de p de 1 à $q-1$ (en prenant $u_0 = 1$). On obtient :

```
q=11
p=1 u(n)=1
p=2 u(n)=1 2 4 8 5 10 9 7 3 6
p=3 u(n)=1 3 9 5 4
p=4 u(n)=1 4 5 9 3
p=5 u(n)=1 5 3 4 9
p=6 u(n)=1 6 3 7 9 10 5 8 4 2
p=7 u(n)=1 7 5 2 3 10 4 6 9 8
p=8 u(n)=1 8 9 6 4 10 3 2 5 7
p=9 u(n)=1 9 4 3 5
p=10 u(n)=1 10
```

ce qui montre que les racines primitives de 11 sont 2, 6, 7 et 8.
Les cas `p=2` à `p=10` sont représentés graphiquement sur la figure suivante :

47. ce qui n'est pas fait ici
48. donc changer de germe ne produit qu'une permutation circulaire des u_n
49. cependant une suite réellement aléatoire de $q-1$ entiers compris entre 1 et $q-1$ aurait une probabilité très faible de ne pas comporter deux valeurs égales

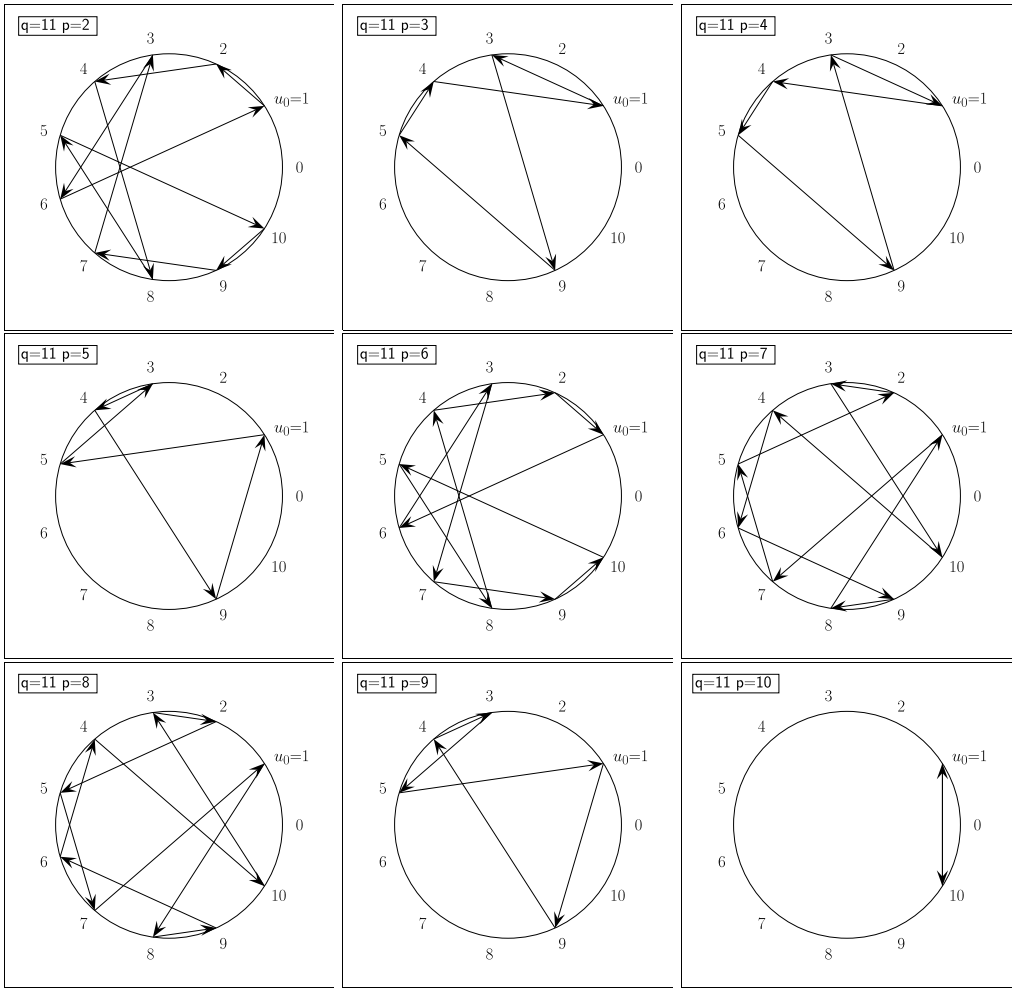


FIGURE 14 –

Pour l'usage pratique qui nous intéresse on veut une séquence d'u_n la plus longue possible. On peut choisir par exemple le nombre premier $q = 1664713$ et la racine primitive $p = 1288$. Il y a un très grand nombre de choix possibles mais q doit être suffisamment grand pour que la période soit grande et on est limité par le fait que le produit $p(q-1)$ ne doit pas dépasser la valeur maximum possible pour un `int` c'est à dire $2^{31}-1$. On a constitué ainsi l'équivalent approché d'un dé à $q-1 = 1664712$ faces équiprobables. Mais ce dé n'en est pas vraiment un parce que si on le lance 1664712 fois on n'obtiendra jamais la même valeur et, au delà de 1664712, on retrouvera la même séquence.

Remarque

La méthode pour simuler le dé à $q-1$ faces utilisée ici est très sommaire. Il en existe de plus raffinées permettant d'avoir des suites « ressemblant mieux » à des suites aléatoires.

2 Simulation d'une fonction à valeurs pseudo-aléatoires réelles à distribution uniforme sur $[0,1]$

On souhaite écrire une fonction qui, à chaque appel, renvoie une valeur aléatoire réelle x telle que la probabilité dp pour qu'elle soit comprise entre x et $x+dx$ est :

$dp = dx$ pour $x \in [0,1]$
$dp = 0$ sinon

Notation :

On désigne par $\text{rect}_{[a,b]}(x)$ la fonction valant 1 pour $a \leq x \leq b$ et 0 ailleurs.

La densité de probabilité de la variable aléatoire x est donc $\text{rect}_{[0,1]}(x)$.
Pour cela, on utilise le dé à $q-1$ faces équiprobables numérotées de 1 à $q-1$ réalisé à la section précédente et, pour un lancer

du dé donnant la valeur n, on prend pour x la valeur réelle $\frac{n-1}{q-2}$.
Cette fonction peut être programmée de la façon suivante :

```
double alea() {
  const int q = 1664713, p = 1288;
  static int n = 1;
  n = n * p % q;
  return (double)(n-1)/(q-2);
}
```

(on a choisi arbitrairement le terme initial égal à 1). Il est indispensable que `n` soit statique sinon elle serait ré-initialisée à 1 à chaque appel de `alea()` qui fournirait toujours la même valeur.
Un inconvénient de cette fonction est qu'à chaque nouvelle exécution du programme qui la contient, la séquence sera la même puisque la valeur initiale est toujours 1. On rajoute donc une seconde fonction permettant de faire varier la valeur initiale et nommée pour cela `germe`, ce qui conduit à :

```
const int q = 1664713, p = 1288;
int n = 1;
void germe(int g) {
  n = g % q; /* on met g%q et non g pour que n*p ne risque pas de depasser
                la valeur max permise pour un int : 2^31-1) */
  if(n == 0) {cout << "Prendre une valeur du germe >= 1 et <= " << q-1 << endl; exit 0;}
}
double alea() {
  n = n * p % q;
  return (double)(n-1)/(q-2);
}
```

la variable globale `n` étant commune aux deux fonctions `germe` et `alea`, on modifie sa valeur en appelant `germe` avec une valeur quelconque de l'argument.
Remarque

Les deux instructions :

```
n = g % q;
if(n == 0) ...
```

peuvent être condensées en une seule :

```
if((n = g%q) == 0) ...
```

ou même :

```
if(!(n = g%q)) ...
```

Exemple d'utilisation de `germe` et `alea` :

```
#include<iostream>
using namespace std;
#include<bibli_fonctions.h>
int main() {
  int i;
  germe(469880);
  for(i = 1; i <= 10; i++)
    cout << alea() << endl;
  return 0;
}
```

Résultat :

0.549417
0.648615
0.415484
0.142915
0.0749385
0.521488
0.676658
0.535218
0.361277
0.325148

Il n'est pas nécessaire de ré-écrire les fonctions `germe` et `alea` pour les utiliser, car elles sont placées dans une bibliothèque écrite au fur et à mesure pour les besoins du magistère. Il suffit de placer un `#include<bibli_fonctions.h>` au début du fichier contenant le programme, et `germe` et `alea` sont alors disponibles, exactement comme les fonctions de la bibliothèque mathématique du C telles que `sin` ou `pow` le sont, à condition d'avoir mis un `#include<math.h>`.
On verra dans un chapitre ultérieur comment un utilisateur peut constituer ainsi sa propre bibliothèque.

La fonction `alea` a des applications nombreuses et puissantes en physique sous le nom de « Méthode de Monte-Carlo » dont on étudie quelques exemples dans la suite.

Remarque

Le C fournit une fonction analogue à `alea` nommée `drand48` qui s'emploie exactement de la même façon. Pour pouvoir l'utiliser il faut mettre `#include<stdlib.h>` (en C) ou `#include<cstdlib>` (en C++). L'équivalent de `germe` est `srand48`. Le programme précédent ré-écrit avec `drand48` et `srand48` se présente ainsi :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i;
  srand48(469880);
  for(i = 1; i <= 10; i++)
    cout << drand48() << endl;
  return 0;
}
```

Résultat :

0.759624
0.717241
0.8973
0.785434
0.366478
0.66685
0.0656149
0.639337
0.41923
0.117331

L'intérêt de `drand48` est qu'il utilise des entiers de 6 octets au lieu de 4 et fournit donc plus de nombres pseudo-aléatoires différents que `alea`. L'intérêt de `alea` est qu'il est particulièrement simple ce qui permet de voir facilement le mécanisme de génération des nombres pseudo-aléatoires. Les générateurs pseudo-aléatoires ayant tous des défauts, il est prudent de faire le même calcul avec plusieurs d'entre eux, aussi différents que possible les uns des autres.

3 Test du générateur uniforme

3.1 Distribution statistique

On a vu dans les sections précédentes comment générer des nombres pseudo-aléatoires sur l'intervalle $[0,1]$ avec une densité uniforme, à l'aide de la fonction `alea()`. On peut tester la fonction `alea()` en constituant la distribution statistique d'un grand nombre de tirages[50]. Pour cela on divise l'intervalle $[0,1]$ en N sous-intervalles consécutifs de longueurs égales :

$[0,1/N[$, $[1/N,2/N[$, $[2/N,3/N[$ ... $[(N-1)/N,1]$

et, pour une suite d'un grand nombre de tirages, on compte combien sont tombés dans le premier intervalle, combien dans le second, etc. jusqu'au $N^{\text{ième}}$ intervalle. On note y_i le nombre de tirages tombés dans l'intervalle numéro i. Les y_i représentent une fonction à valeurs entières y de la variable entière i qui est la distribution statistique sur N intervalles des valeurs successives de `alea()`. Ceci peut, schématiquement, être fait par les instructions suivantes :

```
int i, k, kmax = 10000, n = 100  // kmax nombre de tirages, n nombre d'intervalles
int* y = (int*)malloc(n*sizeof(int)); // (ou int* y = I_1(n); si on utilise
...                                   // les fonctions du Magistère)
for(k = 1; k <= kmax; k++) {
  i = floor(alea()*n);
  if(i < 0 || i > n-1) continue;      // on teste i pour être sûr de ne
```

50. ici : tirage = valeur obtenue par un appel de `alea()`

```
    y[i] = y[i]+1;                    // pas sortir des limites du tableau
  }
  ...
```

Il ne reste plus ensuite qu'à faire tracer la fonction y_i sous forme d'histogramme par exemple, et on obtient :

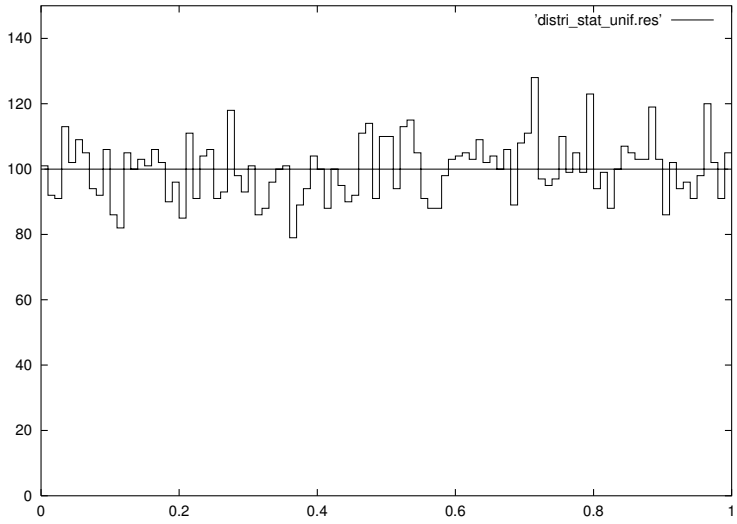


FIGURE 15 –

3.2 Corrélations

On peut également tester s'il y a corrélation entre deux tirages consécutifs x_n et x_{n+1} en portant dans un plan les points de coordonnées $x = x_n$ et $y = x_{n+1}$. On obtient :

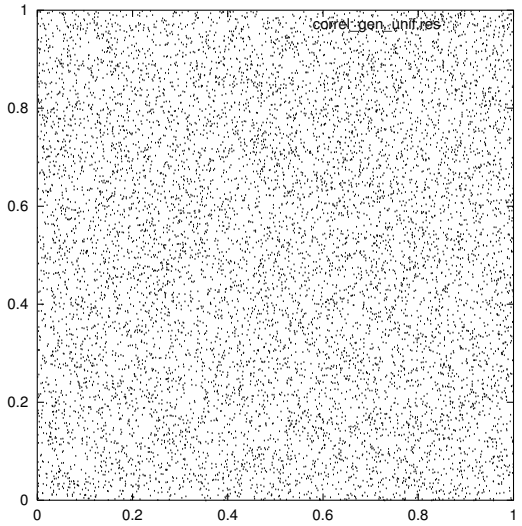


FIGURE 16 –

4 Générateur d'une variable aléatoire suivant une loi donnée

On remarque que si x est une variable aléatoire de densité $\text{rect}_{[0,1]}(x)$, $px+q$ est une v.a. de densité $\dfrac{1}{p}\text{rect}_{[q,p+q]}(x)$ (le facteur $\frac{1}{p}$ assurant la normalisation à 1). De façon plus générale, comment, à partir d'une v. a. de densité uniforme, construire une v. a. de densité quelconque ?

4.1 Méthode de l'inversion de la primitive

4.1.1 Principe

1) Soit x une v. a. de densité $f(x)$. Par définition de la densité, la probabilité dp pour que x soit dans l'intervalle $[x, x+dx]$ est :

$$dp = f(x)\,dx$$

Soit une fonction $y(x)$: quelle est la densité $g(y)$ de la v. a. y ? On a :

$$x \in [x, x+dx] \Longrightarrow y \in [y, y+dy]$$

donc, par définition de la densité de y :

$$dp = g(y)\,dy = f(x)\,dx \Longrightarrow g(y) = f(x)\frac{dx}{dy}$$

2) On envisage la réciproque du problème précédent : quelle doit être la fonction $y(x)$ pour que, x ayant la densité $f(x)$, y ait la densité $g(y)$? On a vu que :

$$g(y)\frac{dy}{dx} = f(x)$$

Soit G une primitive de g, F une primitive de f :

$$\frac{d}{dx}G(y) = g(y)\frac{dy}{dx} = f(x) = \frac{dF}{dx} \Longrightarrow G(y) = F(x) + \text{cte}$$

On se place dans le cas où G^{-1} existe et est calculable[51]. On peut alors écrire :

$$y = G^{-1}\left[F(x) + \text{cte}\right]$$

Il est normal de trouver une constante arbitraire puisqu'on a résolu une équation différentielle du premier ordre.
Le choix de la valeur y_0 que l'on veut attribuer à y pour une valeur particulière de x notée x_0 détermine la constante qui vaut : $G(y_0) - F(x_0)$. Finalement :

$$\boxed{y = G^{-1}\left[F(x) - F(x_0) + G(y_0)\right]}$$

Cas particulier :

si $f(x) = \text{rect}_{[0,1]}(x)$ une primitive de $f(x)$ est $F(x) = x$ et :

$$y = G^{-1}\left[x - x_0 + G(y_0)\right]$$

4.1.2 Exemple

Prenons :

$$f(x) = \text{rect}_{[0,1]}(x)$$

$$g(y) = \frac{1}{\alpha}\exp(-\frac{y}{\alpha}) \text{ pour } y \geq 0 \text{ et } 0 \text{ ailleurs } (\alpha > 0)$$

$g(y)$ est la loi exponentielle que l'on rencontre fréquemment en physique.
α est un paramètre qui représente la valeur moyenne de y :

$$< y >= \frac{1}{\alpha}\int_0^{+\infty} y\exp(-\frac{y}{\alpha})dy = \alpha$$

Une primitive de $g(y)$ est $G(y) = -\exp(-\frac{y}{\alpha})$ dont la fonction réciproque est $G^{-1}(z) = -\alpha\ln(-z)$. Donc :

$$y = G^{-1}\left[x - x_0 + G(y_0)\right] = -\alpha\ln\left[-x + x_0 + \exp(-\frac{y_0}{\alpha})\right]$$

Choisissons par exemple $y(0) = 0$, on a alors $x_0 = 0$ et $y_0 = 0$ et :

$$y = -\alpha\ln(-x+1)$$

$1 - x$ et x ayant la même distribution $\text{rect}_{[0,1]}(x)$ on peut écrire plus simplement :

$$\boxed{y = -\alpha\ln(x)}$$

On pourra donc, dans un programme, tirer des valeurs aléatoires dont la densité suit une loi exponentielle de paramètre α par l'instruction :

51. Sinon il faudra employer la méthode de Von Neuman exposée ci-dessous

```
...
y = -alpha*log(alea());
...
```

On peut, exactement comme dans le cas uniforme, faire une distribution statistique des valeurs obtenues avec, par exemple, 10000 tirages et comparer avec la courbe théorique $1/\alpha \exp(-y/\alpha)$, ce qui donne le résultat suivant :

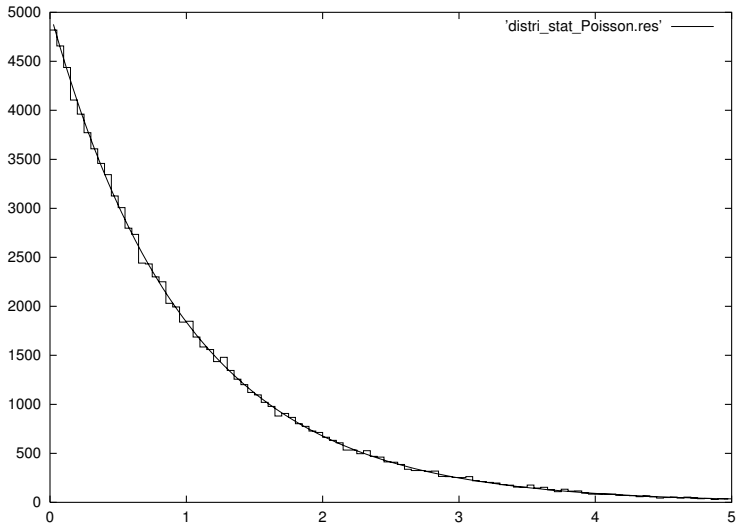


FIGURE 17 –

4.2 Méthode du rejet de Von Neuman

Dans le cas où la primitive de la densité recherchée n'est pas inversible et calculable, on applique une méthode beaucoup plus générale.

Soit x une v. a. de loi $f(x)$ sur $[a, b]$ et soit M la valeur supérieure[52] de $f(x)$ sur $[a, b]$.

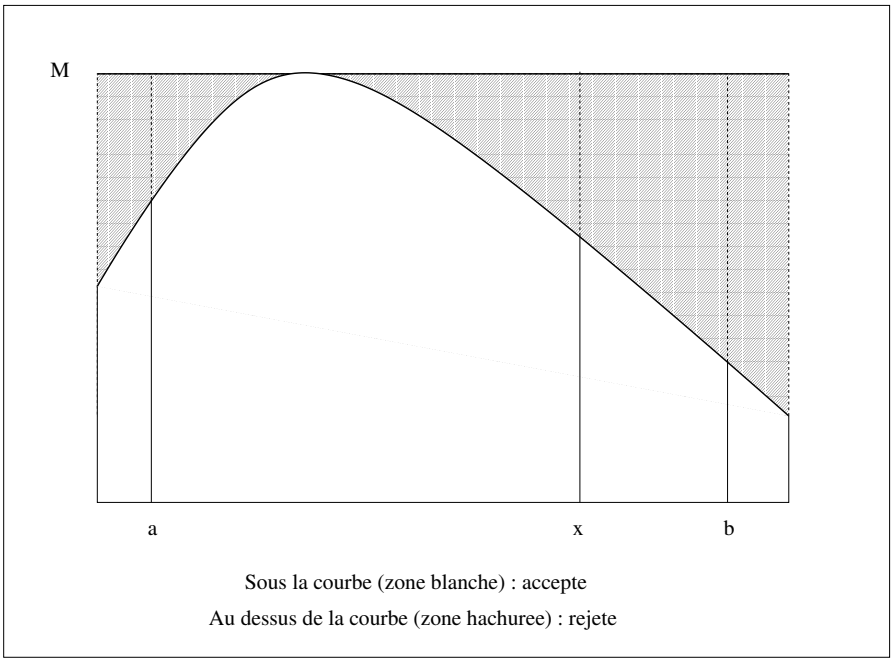


FIGURE 18 –

52. supposée exister

On tire une valeur aléatoire x avec une densité uniforme sur $[a, b]$, puis une valeur aléatoire y avec une densité uniforme sur $[0, M]$. Si le point de coordonnées (x, y) est au-dessus de la courbe de $f(x)$ on le rejette, sinon on l'accepte. Les valeurs de x ainsi sélectionnées suivent la loi de densité $f(x)$. En effet, supposons qu'on ait effectué N_0 doubles tirages (x et y). Le nombre de valeurs de x situées dans l'intervalle $[x, x + dx]$ est :

$$dN = N_0 \underbrace{\frac{dx}{b-a}}_{1^{\text{er}}\text{ tirage}} \underbrace{\frac{f(x)}{M}}_{2^{\text{ème}}\text{ tirage}}$$

Pour déduire de dN la densité de probabilité de x, il faut diviser dN par le nombre N de valeurs de x effectivement obtenues, qui est différent de N_0 puisqu'il y a des valeurs rejetées. Le nombre de valeurs de x retenues est :

$$N = \int_a^b dN = \frac{N_0}{(b-a)M} \int_a^b f(x)\, dx$$

$f(x)$ étant une densité de probabilité on a :

$$\int_a^b f(x)\, dx = 1$$

soit :

$$N = \frac{N_0}{(b-a)M}$$

Donc :

$$dp = \frac{dN}{N} = f(x)\, dx$$

La densité de probabilité de x est bien $f(x)$.

Exemple

On veut générer des valeurs aléatoires avec la densité :

$$\frac{3}{2} \sin x \cos^2 x \qquad \text{sur} \qquad [0, \pi]$$

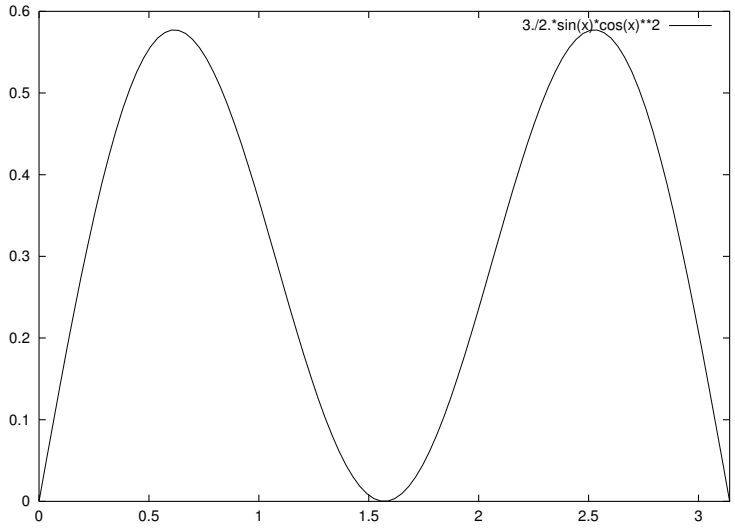


FIGURE 19 –

Le maximum de cette fonction est $1/\sqrt{3}$. Donc :

on tire avec une densité uniforme sur $[0, \pi]$ ce qui donne une valeur x
on tire avec une densité uniforme sur $[0, 1/\sqrt{3}]$ ce qui donne une valeur y

si $y \leq \frac{3}{2} \sin x \cos^2 x$ alors on retient x sinon on le rejette et on recommence.

Remarque

En pratique si, au lieu d'utiliser $f(x)$ qui est normée à 1, on utilise $f'(x) = \lambda f(x)$ (λ quelconque $\in \mathbb{R}$), le résultat est le même puisque le maximum M' de $f'(x)$ est multiplié dans le même rapport λ et on a : $f(x)/M = f'(x)/M'$. Il n'est donc pas nécessaire de travailler avec une fonction normée à 1.

Complément :

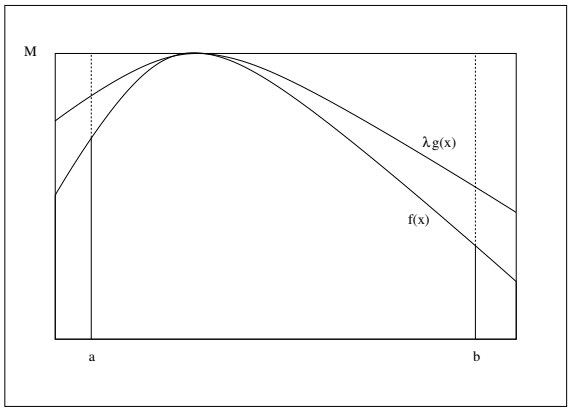


FIGURE 20 –

L'inconvénient de cette méthode est qu'il peut y avoir beaucoup de tirages inutiles. On peut y remédier si on connaît une densité $g(x)$ sur $[a,b]$ pour laquelle on sait générer des valeurs aléatoires et que cette densité satisfait à la propriété :

$$\exists \lambda \in \mathbf{R} \text{ tel que } \lambda g(x) \geq f(x)\ \forall x \in [a,b]$$

On procède encore à un double tirage mais :

au lieu de tirer x avec une densité uniforme sur $[a,b]$, on le tire avec la densité $g(x)$
on tire y avec une densité uniforme sur $[0, \lambda g(x)]$

puis, comme précédemment, on rejette le point s'il est au-dessus de la courbe de $f(x)$.

Pour N_0 doubles tirages le nombre de valeurs de x situées dans l'intervalle $[x, x+dx]$ est donc :

$$dN = N_0 \underbrace{g(x)dx}_{1^{\text{er}}\text{ tirage}} \underbrace{\frac{f(x)}{\lambda g(x)}}_{2^{\text{ème}}\text{ tirage}}$$

Le nombre de valeurs de x retenues est :

$$N = \int_a^b dN = \frac{N_0}{\lambda} \int_a^b f(x)\,dx = \frac{N_0}{\lambda}$$

Donc :

$$dp = \frac{dN}{N} = f(x)\,dx$$

La densité de probabilité de x est bien $f(x)$.

Remarque

On est obligé d'introduire le facteur λ car on ne pourrait avoir simultanément : $\int_a^b f(x)\,dx = 1$, $\int_a^b g(x)\,dx = 1$ et $g \geq f$ sauf si $f = g$.

5 Applications

5.1 Mouvement brownien

On étudie le mouvement brownien d'une particule dans un plan. Entre deux chocs contre les molécules du milieu elle parcourt en ligne droite une distance l dont la densité de probabilité est une loi exponentielle dont le paramètre est son libre parcours moyen $\bar{l}$. Chaque choc est considéré comme le tirage au hasard :

d'un angle entre 0 et 2π avec une densité uniforme
d'une longueur entre 0 et l'infini avec une densité exponentielle

On a donc déjà tous les éléments pour faire le calcul et on obtient :

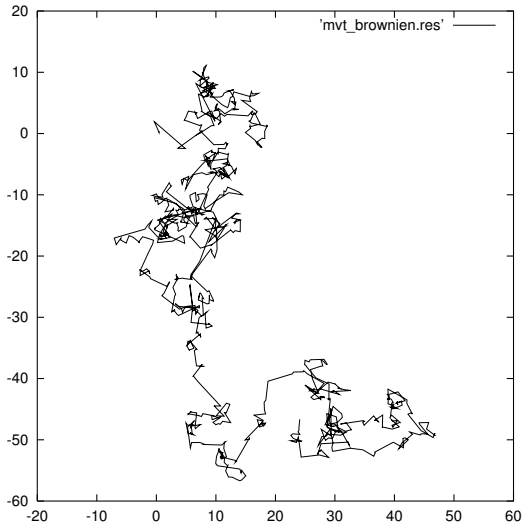


FIGURE 21 –

À partir de là, en étudiant les positions d'un grand nombre de particules en fonction du nombre de chocs subis, on reproduit facilement la diffusion d'un nuage de particules dans un gaz par exemple (diffusion de la fumée de cigarette en l'absence de courant d'air).

5.2 Rayonnement d'une sphère radioactive

On considère une sphère homogène, de rayon R, constituée d'un matériau radioactif émettant des photons γ. Un photon peut être émis depuis n'importe quel point de la sphère et dans n'importe quelle direction avec la même probabilité. On peut considérer schématiquement que dans la matière, chaque photon est absorbé après un trajet rectiligne dont la longueur l varie aléatoirement avec la densité :

$$\frac{1}{\alpha}\exp(-\frac{l}{\alpha}) \qquad (1)$$

Les parcours des photons peuvent être représentés de la façon suivante :

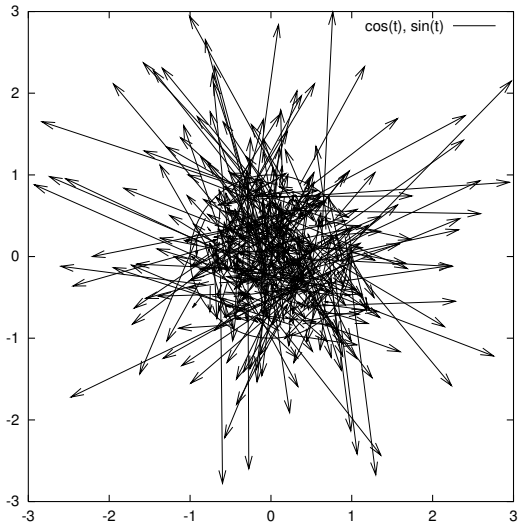


FIGURE 22 –

(contrairement à ce qu'indique la figure, les photons qui sortent de la sphère ne sont plus absorbés ensuite)

On sait que α représente le libre parcours moyen (=paramètre du problème).
On veut calculer le taux moyen τ de photons qui sortent de la sphère :

$$\tau = \frac{\text{nombre de photons sortant}}{\text{nombre de photons émis depuis l'intérieur de la sphère}}$$

Il est évident que τ dépend de la valeur relative de $\langle l \rangle$ et R et on peut dire à priori :

$\langle l \rangle >> R \implies \tau \simeq 1$
$\langle l \rangle << R \implies \tau \simeq 0$

on cherche donc à calculer τ en fonction du rapport $\langle l \rangle / R$.

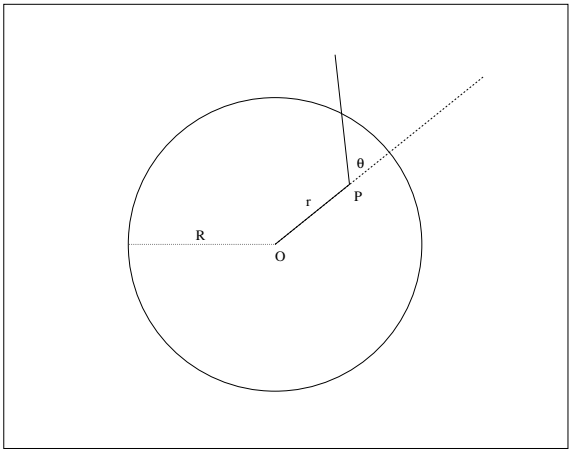


FIGURE 23 –

Pour cela on simule l'expérience :

on tire au hasard un noyau émetteur P dans la sphère et une direction d'émission.
on en déduit la longueur d à parcourir pour, à partir de ce point et dans cette direction, parvenir à la surface de la sphère
on tire au hasard une longueur de parcours l du photon selon la loi (1)
si $l \geq d$ le photon sort

On renouvelle cette simulation un grand nombre de fois et on en déduit τ.

La sphère étant homogène et l'émission isotrope le problème se simplifie. Tous les noyaux émetteurs situés à la même distance du centre de la sphère donnent la même contribution. Pour tirer au hasard la position d'un noyau il suffit de tirer une valeur de r entre 0 et R avec un densité proportionnelle à l'élément de volume $4\pi r^2\, dr$.
On repère la direction d'émission par ses angles (θ, φ) dans un repère sphérique de centre P et d'axe $Oz \parallel OP$. Le système constitué par la sphère et le noyau ayant la symétrie de révolution autour de l'axe OP, la valeur de φ n'intervient pas. Il suffit de tirer un angle θ entre 0 et π avec une densité proportionnelle à l'élément d'angle solide $2\pi \sin\theta\, d\theta$.
Enfin un calcul géométrique indique que la longueur à parcourir pour sortir de la sphère depuis le point P dans la direction déterminée par θ est :

$$d = -r\cos\theta + \sqrt{R^2 - r^2\sin^2\theta}$$

Les densités de r et θ sont assez simples pour qu'on puisse générer les valeurs aléatoires par la méthode de l'inversion de la primitive.

Génération des valeurs de r

La densité de r est :

$$\frac{4\pi r^2\, dr}{\frac{4}{3}\pi R^3} = \frac{3r^2\, dr}{R^3}$$

x étant la v. a. à densité uniforme sur $[0, 1]$ il s'agit de trouver la fonction $r(x)$ telle que r ait la distribution précédente. On a vu que :

$$r = G^{-1}\left[x - x_0 + G(r_0)\right]$$

G étant ici une primitive de $\dfrac{3r^2}{R^3}$ c'est à dire $\dfrac{r^3}{R^3}$. G est donc inversible et on a :

$$r = R\left(x - x_0 + \frac{r_0^3}{R^3}\right)^{1/3}$$

On choisit que $r = 0$ pour $x = 0$, on a donc finalement :

$$r = Rx^{1/3}$$

Génération des valeurs de θ

La densité de θ est :

$$\frac{2\pi \sin\theta}{4\pi} = \frac{\sin\theta}{2}$$

donc ici :

$$G = -\frac{\cos\theta}{2}$$

Cette fonction est inversible sur $[0, \pi]$:

$$\theta = \arccos\left[-2(x - x_0 - \frac{1}{2}\cos\theta_0)\right]$$

On choisit que $\theta = 0$ pour $x = 0$, donc :

$$\theta = \arccos(-2x + 1)$$

Génération des valeurs de l

On a vu ci-dessus que, la densité de l étant une loi exponentielle de paramètre $\alpha = \langle l \rangle$:

$$l = -\alpha \ln x$$

Tous les éléments étant réunis, la simulation d'une émission peut s'écrire :

```
...
double a = 1./3.;
for(s = 0, i = 1; i <= n; i++) {
  r = pow(alea(),a)*R;                  /* position du noyau emetteur */
  th = acos(2.*alea()-1.);              /* direction d'emission du photon */
  rs = r*sin(th);
  d = -r*cos(th) + sqrt(R2-rs*rs);      /* distance a parcourir pour sortir */
  do
    x = alea();
  while(x == 0.);
  l = -alpha*log(x);                    /* longueur parcourue par le photon */
  if(l > d)
    s++;                                /* on compte les photons qui sortent */
}
...
```

Le programme entier est donné en annexe. Il fait ce calcul pour des valeurs de α échelonnées entre 0 et R, ce qui donne la courbe suivante pour τ en fonction de α/R :

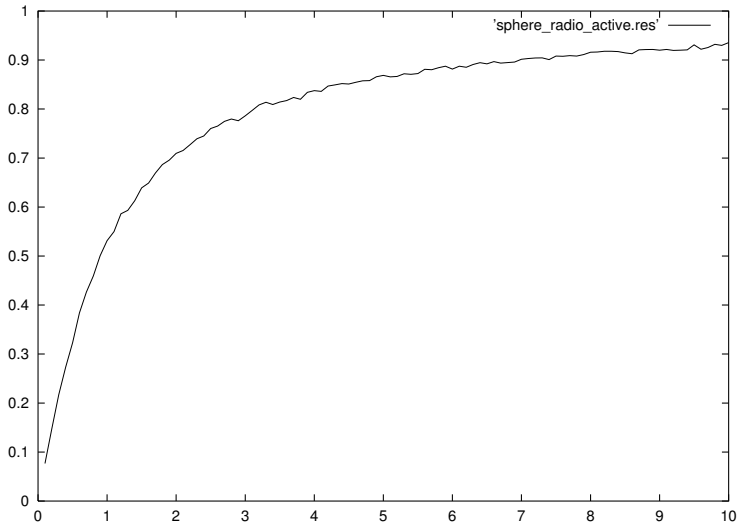


FIGURE 24 –

Remarque

Pour faire le calcul analytiquement il faudrait calculer l'intégrale :

$$8\pi^2 \int_0^R \int_0^\pi r^2 \sin\theta_P \left[1 - \exp(-\frac{d}{\alpha})\right] \, dr \, d\theta_P$$

avec :

$$d = -r\cos\theta + \sqrt{R^2 - r^2 \sin^2\theta}$$

5.3 Miroir hémisphérique

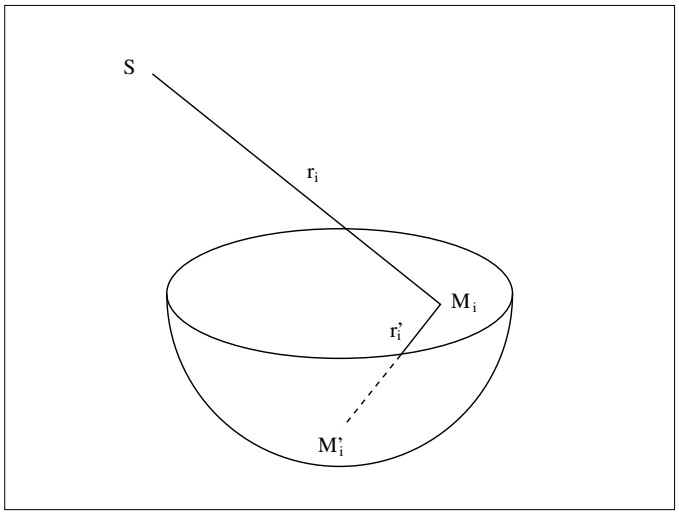


FIGURE 25 –

On considère un miroir hémisphérique concave et une source ponctuelle S, située n'importe où, mais de telle façon qu'au moins une partie de ses rayons puissent se réfléchir sur la surface concave du miroir [53]. On tire une séquence de rayons lumineux r_i issus de S, répartis au hasard de façon isotrope dans toutes les directions de l'espace. Si un rayon

53. S doit donc être du côté opposé à la demi-sphère par rapport au plan Π contenant le bord du miroir

r_i atteint la surface concave du miroir on appelle M_i le point d'incidence et on calcule la position de l'intersection M_i' du premier rayon réfléchi r_i' avec le miroir. Puis on représente graphiquement en perspective le miroir et l'ensemble des points M_i'. On obtient la caustique suivante :

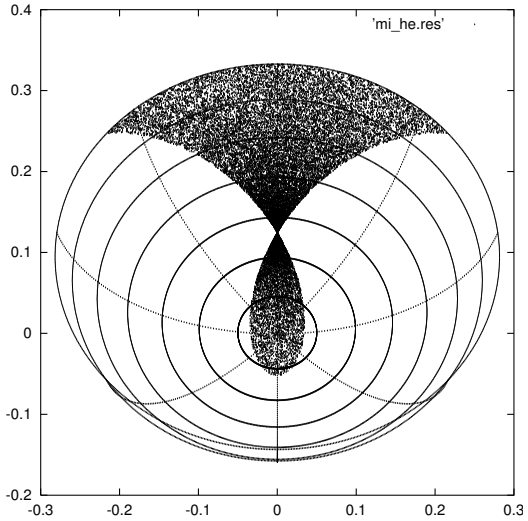


FIGURE 26 –

La méthode consistant à utiliser des rayons lumineux distribués au hasard de façon isotrope est la plus simple car il n'est pas possible de répartir de façon régulière un grand nombre de rayons dans toutes les directions de l'espace.

5.4 Représentation d'une orbitale atomique

5.4.1 Densité de probabilité de présence

Considérons l'état $n = 2$, $l = 1$, $m = 0$ de l'électron de l'atome d'hydrogène. La fonction d'onde s'écrit :

$$\psi(\mathbf{r}) = \frac{1}{4\sqrt{2\pi a_0^3}} \frac{r}{a_0} \exp\left(-\frac{r}{2a_0}\right) \cos\theta$$

(les vecteurs sont représentés en caractères gras)
a_0 = rayon de Bohr $= \dfrac{\hbar^2}{\mu e^2}$ =0.52 Å, μ = masse réduite du système électron proton, e = charge de l'électron.
La densité de probabilité de présence de l'électron dans l'élément de volume $d^3\mathbf{r} = r^2 \sin\theta\, dr\, d\theta\, d\varphi$ est :

$$|\psi(\mathbf{r})|^2 = \alpha \left(\frac{r}{a_0}\right)^2 \exp\left(-\frac{r}{a_0}\right) \cos^2\theta$$

α est une constante qu'il est inutile d'expliciter. On choisit a_0 comme unité de longueur. La probabilité de présence dp dans l'élément de volume $d^3\mathbf{r}$ est donc :

$$dp = \alpha r^4 \exp(-r) \sin\theta \cos^2\theta\, dr\, d\theta\, d\varphi$$

et, puisque cette densité ne dépend pas de φ, il suffit de la représenter dans un plan φ =constante[54] en fonction de r et θ. D'autre part cette densité est un produit d'une fonction de r par une fonction de θ donc l'étude, à priori à deux dimensions, se réduit à deux fois une dimension. Il suffit de générer des couples r, θ avec la loi :

$r^4 \exp(-r)$ pour $r \in [0, +\infty[$
$\sin\theta \cos^2\theta$ pour $\theta \in [0, \pi]$

On peut de plus utiliser le fait que la fonction $\sin\theta \cos^2\theta$ est symétrique par rapport à $\pi/2$ et se contenter de tirer θ dans l'intervalle $[0, \pi/2]$.

54. c'est à dire un plan passant par Oz

5.4.2 Loi de r

La fonction $f(r) = r^4 \exp(-r)$ peut être considérée comme négligeable à partir d'une certaine valeur $r_{\max}$ de r.

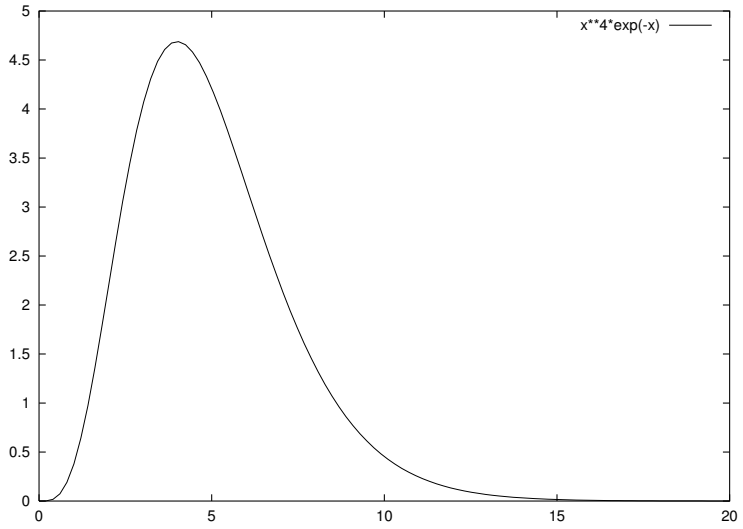


FIGURE 27 –

En prenant par exemple $r_{\max} = 15$ on a $f(r_{\max}) = 3.3\,10^{-4}$.
Par ailleurs la valeur maximum atteinte par $f(r)$ est $M = (4/e)^4$. Finalement :

on tire avec une densité uniforme sur $[0, r_{\max}]$ ce qui donne une valeur r
on tire avec une densité uniforme sur $[0, M]$ ce qui donne une valeur x
si $x \leq f(r)$ on garde r sinon on le rejette.

5.4.3 Loi de θ

A la section **Méthode du rejet de Von Neuman** on a déjà vu, en exemple, comment générer des valeurs de θ suivant la loi $\sin\theta\cos^2\theta$.

5.4.4 Mise en œuvre

Le cœur du programme s'écrit :

```
...
int imax = 30000; double rmax = 15.;
const double c1 = pow(4./exp(1.),4.), c2 = 2./3./sqrt(3.);
...
for(i = 1; i <= imax; i++) {
  do {
    r = rmax*alea();
    x = c1*alea();
  } while(x > pow(r,4.)*exp(-r));
  do {
    tet = 2.*M_PI*alea();
    x = c2*alea();
  } while(x > fabs(sin(tet)*cos(tet)*cos(tet)));   /* on doit prendre la valeur absolue
                                                      parce qu'on choisit de tirer tet sur
                                                      [0,2pi] au lieu de [0,pi/2] */
  fich << r*cos(tet) << " " << r*sin(tet) << endl;
  }
...
```

ce qui donne la représentation :

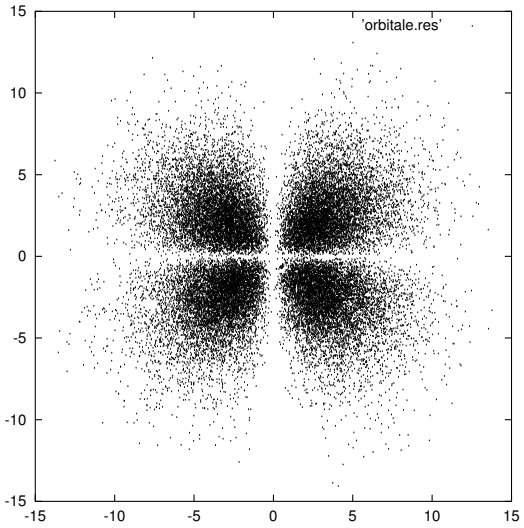


FIGURE 28 –

Remarque

On a choisi de tirer θ sur $[0, 2\pi]$ au lieu de $[0, \pi/2]$ uniquement pour obtenir une représentation sur les quatre quadrants, donc plus concrète.

A titre de comparaison, le tracé de la surface $z = r^4 \exp(-r) \sin\theta \cos^2\theta$ donne :

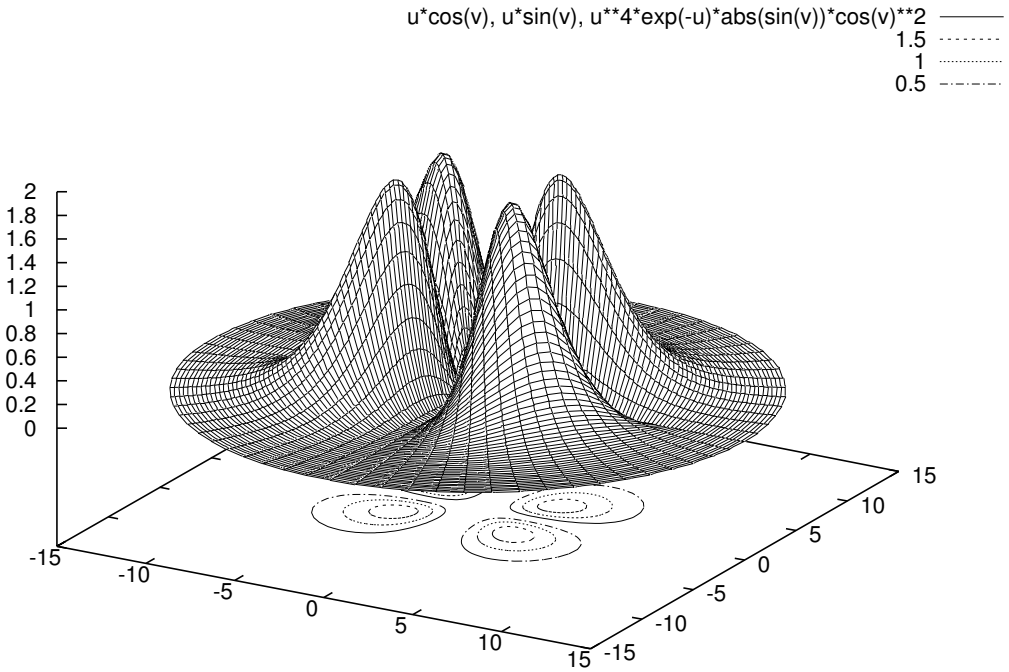


FIGURE 29 –

Annexe : rayonnement d'une sphère radioactive

```
#include<iostream>
#include<fstream>
#include<math.h>
#include<stdlib.h>
#include<time.h>
using namespace std;
int main() {
/* CALCUL DU TAUX DE SORTIE DE PHOTONS GAMMA D'UNE SPHERE RADIOACTIVE
   EN FONCTION DU RAPPORT: PARCOURS MOYEN DES PHOTONS/RAYON DE LA SPHERE
   Notations:
   R rayon de la sphere
   alpha parcours moyen des photons dans la sphere
   r distance du noyau emetteur P au centre O de la sphere
   th angle entre OP et la direction d'emission du photon
   d distance a parcourir pour sortir
   l longueur parcourue par le photon
   s compteur de sorties
   n nombre d'emissions simulees
   k indice de la boucle sur les differentes valeurs de alpha/r
   kmax nombre de valeurs de alpha/r
   i indice de la boucle sur les n emissions simulees
   x variable aleatoire uniforme sur [0,1], de probabilite nulle en dehors
   ********************************************************************************
  -----Declarations----- */
  fstream fich;
  double l, R = 1., R2 = R*R, rs, alpha, d, x, r, th;
  int i, n = 10000, s, k, kmax = 100;
  /* ---------------------- */
  srand48(time(NULL));  /* Initialise les nombres aleatoires par l'heure actuelle en secondes */
  fich.open("sphere_radioactive.res", ios::out);
  for(k = 1; k <= kmax; k++) {
    alpha = k*R/10.;
    for(s = 0, i = 1; i <= n; i++) {
      r = pow(drand48(), 1./3.) * R;       /* position du noyau emetteur */
      th = acos(2.*drand48()-1.);          /* direction d'emission du photon */
      rs = r*sin(th);
      d = -r*cos(th) + sqrt(R2-rs*rs);     /* distance a parcourir pour sortir */
      do
        x = drand48();
      while(x == 0.);
      l = -alpha*log(x);                   /* longueur parcourue par le photon */
      if(l > d)
        s++;
    }
    cout << "k = " << k << endl;
    cout << "nb de des. = " << n << "   nb de photons sortis = " << s
         << "   taux de sortie = " << (double)s/n << endl;
    fich << alpha << " " << (double)s/n << endl;
  }
  fich.close();
  return 0;
}
```

Annexe : représentation d'une orbitale atomique

```
#include<fstream>
#include<math.h>
#include<stdlib.h>
#include<time.h>
using namespace std;
int main() {
  double r, tet, x, rmax;
  const double c1 = pow(4./exp(1.),4.), c2 = 2./3./sqrt(3.);
  int i, imax;
  fstream fich;
  fich.open("orbitale.res", ios::out);
  imax = 30000; rmax = 15.;
  srand48(time(NULL));
  for(i = 1; i <= imax; i++) {
    do {
      r = rmax*drand48();
      x = c1*drand48();
    } while(x > pow(r,4.)*exp(-r));
    do {
      tet = 2.*M_PI*drand48();
      x = c2*drand48();
    } while(x > fabs(sin(tet)*cos(tet)*cos(tet)));
    fich << r*cos(tet) << " " << r*sin(tet) << endl;
  }
  return 0;
}
```

11. Introduction à la programmation de la résolution numérique d'équations différentielles par les méthodes d'Euler et de Runge-Kutta

1 Équation différentielle du premier ordre

Soit l'équation différentielle suivante :

$$\dot{q} = f(q,t)$$

q étant la fonction inconnue de la variable t, $\dot{q}$ sa dérivée par rapport à t. f est une fonction donnée quelconque. C'est donc une équation de forme assez générale, mais il faut que $\dot{q}$ soit explicitement exprimée en fonction de t et q. C'est une équation du premier ordre, donc la donnée de la valeur $q_0 = q(t_0)$ de la fonction $q(t)$ pour une valeur donnée t_0 de la variable t détermine une solution unique.

1.1 Méthode d'Euler

La méthode la plus simple pour calculer numériquement une solution est celle d'Euler. Au premier ordre :

$$q(t+dt) = q(t) + dt\ \dot{q}(t) = q(t) + dt\ f\left[q(t),t\right]$$

expression qui permet de calculer une valeur approchée de $q(t+dt)$ à partir de la donnée de $q(t)$, cette valeur approchée étant d'autant plus exacte que le pas dt est plus petit. On calcule donc $q(t_0+dt)$ à partir de $q(t_0)$, puis en réitérant $q(t_0+2dt)$ à partir de $q(t_0+dt)$, et ainsi de suite (d'où le nom de méthode pas à pas) et on obtient une suite de valeurs approchées de la solution unique de l'équation initiale.

1.2 Programmation de la formule d'Euler

Prenons comme exemple l'équation différentielle $\dot{q} = -t\ q$ avec les conditions aux limites $t_0 = 0$, $q_0 = 1$, dont la solution est $\exp(-t^2/2)$. Évidemment le calcul approché par une méthode numérique de la solution n'est intéressant que dans les cas où on ne connaît pas la solution mathématique. Mais ici la connaissance de cette solution nous permet de vérifier nos calculs.

1.2.1 Programme le plus simple

```
// euler_1.cpp
#include<bibli_fonctions.h>
int main() {
  int i, np;
  double q, qp, t, dt, tfin;
  fstream res;
  res.open("euler_1.res", ios::out);  // fichier pour écrire les résultats
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                       // conditions initiales
  for(i = 1; i <= np; i++) {          // boucle sur la variable t
    res << t << " " << q << endl;
    qp = -t*q;                        // équation différentielle
    q = q + dt*qp;                    // calcul d'Euler
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_1.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
```

```
        << "plot(x, y, '+')\n"
        << "plot(x, exp(-x*x/2.))\n"
        ;
    make_plot_py(pyth);
    return 0;
}
```

où :

`tfin` est la valeur finale choisie pour t (ici 3)
`np` est le nombre de points que l'on veut calculer, y compris les valeurs finale et initiale (ici 30)
`dt` le pas en t ($dt = t_{\text{fin}}/(np-1)$)

La figure suivante montre le résultat obtenu (croix) comparé à la solution exacte (ligne continue).

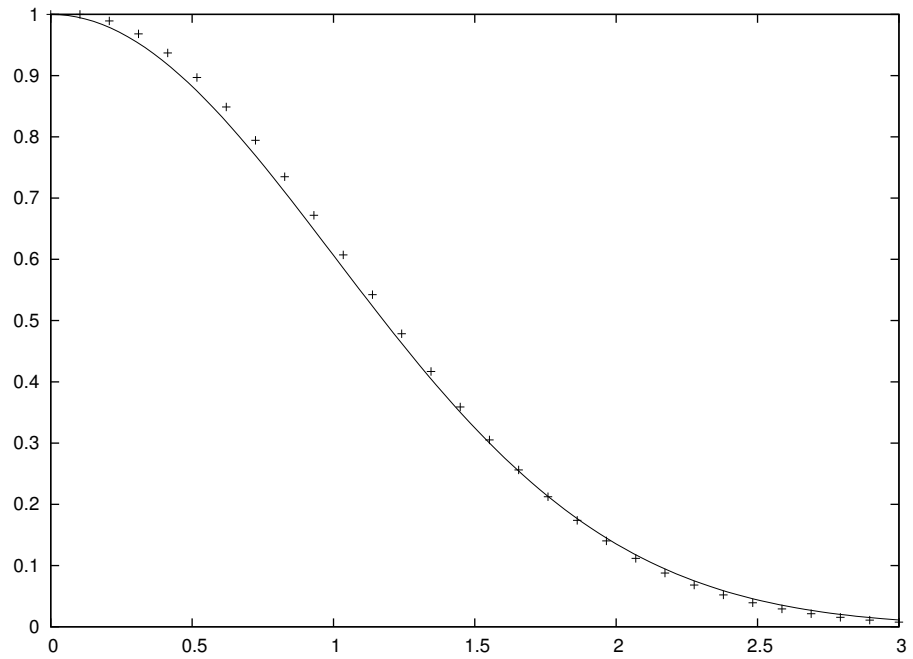


FIGURE 30 –

À partir de ce programme nous faisons maintenant une suite d'améliorations et de généralisations.

1.2.2 On met à part dans une fonction l'expression de l'équation différentielle

Cela isole l'équation différentielle du reste du programme ce qui rend ce dernier plus modulaire donc plus lisible et susceptible d'être modifié plus facilement et avec moins de risques d'erreur.

```
// euler_2.cpp
// On met l'équation différentielle dans une fonction à part
#include<bibli_fonctions.h>
//------------Fonction équation différentielle----------------
double ed(double q, double t) {
  return -t*q;
}
//---------------------------Main-----------------------------
int main() {
  int i, np;
  double q, qp, t, dt, tfin;
  fstream res;
```

```
  res.open("euler_2.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                          // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    qp = ed(q,t);
    q = q + dt*qp;                       // calcul d'Euler
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_2.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
    << "plot(x, y, '+')\n"
    << "plot(x, exp(-x*x/2.))\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

1.2.3 On met à part dans une fonction le calcul d'Euler

Cela isole le calcul d'Euler du reste du programme avec les mêmes avantages que pour l'équation différentielle.

```
// euler_3.cpp
// On met le calcul d'Euler dans une fonction à part
#include<bibli_fonctions.h>
//---------------Fonction équation différentielle--------------
double ed(double q, double t) {
  return -t*q;
}
//----------------------Fonction Euler-------------------------
double euler(double q, double t, double dt) {
  double qp;
  qp = ed(q,t);
  return q + dt*qp;
}
//-----------------------------Main-----------------------------
int main() {
  int i, np;
  double q, t, dt, tfin;
  fstream res;
  res.open("euler_3.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  t = 0; q = 1;                          // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(q,t,dt);
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
```

```
      << "A = loadtxt('euler_3.res')\n"
      << "x = A[:,0]\n"
      << "y = A[:,1]\n"
      << "plot(x, y, '+')\n"
      << "plot(x, exp(-x*x/2.))\n"
      ;
    make_plot_py(pyth);
    return 0;
}
```

Remarque

La variable qp ne figure plus dans le programmme principal, elle est déclarée dans la fonction `euler`.

1.2.4 On met la fonction qui représente l'équation différentielle en argument de la fonction `euler`

Cela permet d'intégrer plusieurs équations différentielles dans une même exécution du programme.

```
// euler_4.cpp
// On met l'équation différentielle à intégrer en argument de la fonction euler
// pour pouvoir intégrer différentes équations différentielles
#include<bibli_fonctions.h>
//--------------Première équation différentielle--------------------
double ed1(double q, double t) {
  return -t*q;
}
//---------------Seconde équation différentielle--------------------
double ed2(double q, double t) {
  return -t*t*q;
}
//-----------------------Fonction Euler-----------------------------
double euler(double(*ed)(double,double), double q, double t, double dt) {
  double qp;
  qp = ed(q,t);
  return q + dt*qp;                    // calcul d'Euler
}
//----------------------------Main----------------------------------
int main()
{
  int i, np;
  double q, t, dt, tfin;
  fstream res;
  res.open("euler_4a.res", ios::out);
  np = 30; tfin = 3; dt = tfin/(np-1);
  // Première équation différentielle
  t = 0; q = 1;                         // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(ed1,q,t,dt);
    t = t+dt;
  }
  res.close();
  res.open("euler_4b.res", ios::out);
  // Seconde équation différentielle
  t = 0; q = 1;                         // conditions initiales
  for(i = 1; i <= np; i++) {
    res << t << " " << q << endl;
    q = euler(ed2,q,t,dt);
```

```
    t = t+dt;
  }
  res.close();
  //-------------------Tracé des courbes----------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_4a.res')\n"
    << "x = A[:,0]\n"
    << "y = A[:,1]\n"
    << "plot(x, y, '+')\n"
    << "plot(x, exp(-x*x/2.))\n"
    << "A = loadtxt('euler_4b.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

2 Système d'équations différentielles du premier ordre

2.1 Méthode d'Euler pour un système

La méthode d'Euler s'applique de la même façon à un système d'équations différentielles du premier ordre. Soit le système différentiel suivant :

$$\begin{aligned} \dot{q}_0 &= f_0(q_0, q_1, \ldots, t) \\ \cdots &= \cdots \\ \dot{q}_i &= f_i(q_0, q_1, \ldots, t) \\ \cdots &= \cdots \\ \dot{q}_{n-1} &= f_{n-1}(q_0, q_1, \ldots, t) \end{aligned}$$

où les q_i sont des fonctions inconnues de la seule variable t, les f_i des fonctions données quelconques, n est la dimension du système[55]. En écrivant que les q_i sont les composantes d'une vecteur $\overrightarrow{q}$ et les f_i celles d'un vecteur $\overrightarrow{f}$ le système s'écrit :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q}, t)$$

Il est du premier ordre, donc la donnée de $\overrightarrow{q}(t_0)$ détermine une solution unique. La méthode d'Euler se généralise à ce système et fournit une valeur approchée de $\overrightarrow{q}(t + dt)$ à partir de $\overrightarrow{q}(t)$:

$$\overrightarrow{q}(t + dt) = \overrightarrow{q}(t) + dt\ \overrightarrow{f}\,[\overrightarrow{q}(t), t]$$

2.2 Programmation de la formule d'Euler pour un système

On considère par exemple le système :

$$\begin{aligned} \dot{q}_0 &= q_1 \\ \dot{q}_1 &= -q_0 \end{aligned}$$

dont la solution ayant pour conditions initiales $q_0(0) = 1$ et $q_1(0) = 0$ est $q_0 = \cos t$, $q_1 = -\sin t$.
On a :

$$\begin{aligned} q_0(t + dt) &= q_0(t) + dt\ q_1(t) \\ q_1(t + dt) &= q_1(t) - dt\ q_0(t) \end{aligned}$$

55. Dans ces équations on fait varier les indices de 0 à $n-1$ plutôt que de 1 à n pour faciliter la transposition en C.

Ici le système ne comprend que deux équations différentielles et on pourrait continuer à utiliser des variables simples `q0` et `q1` par exemple pour mettre les valeurs de q_0 et q_1. Mais lorsque le nombre d'équations est plus grand il devient indispensable d'utiliser un tableau pour mettre les valeurs des q_i. De même pour les $\dot{q}_i$. Nous utilisons donc des tableaux pour que le programme puisse traiter le cas d'un système comportant un nombre quelconque n d'équations.

```
// euler_5.cpp
// On intègre un système d'équations différentielles du premier ordre
#include<bibli_fonctions.h>
//-----------------------Fonction système d'équations différentielles-------------------------
void sda(double* q, double t, double* qp, int n) {
  qp[0] = q[1];
  qp[1] = -q[0];
}
//---------------------------------------Fonction euler-----------------------------------------
void euler(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i;
  double* qp = (double*)malloc(n*sizeof(double)); // ou bien double* qp = D_1(n) si on utilise
  sd(q,t,qp,n);                                   // les fonctions du Magistère
  for(i = 0; i < n; i++)
    q[i] = q[i] + dt*qp[i];                       // calcul d'Euler
  free(qp);   // très important ici parce qu'Euler est appelée un très grand nombre de fois
              // (ou bien f_D_1(qp,n) si on utilise des fonctions du Magistère)
}
//---------------------------------------Main---------------------------------------------------
int main() {
  int i, np, n = 2;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double)); // ou bien double *q=D_1(n) si on utilise
  fstream res;                                   // les fonctions du Magistère
  res.open("euler_5.res", ios::out);
  np = 100; tfin = 7; dt = tfin/(np-1);          // valeurs des paramètres
  t = 0; q[0] = 1; q[1] = 0;                     // conditions initiales
  for(i = 1; i <= np; i++) {                     // boucle sur t
    res << t << " " << q[0] << " " << q[1] << endl;
    euler(sda,q,t,dt,n);
    //rk4(sda,q,t,dt,n);  // seule ligne à changer si on fait Runge-Kutta au lieu d'Euler
    t = t+dt;             // (voir ci-dessous la présentation de la méthode de Runge-Kutta)
  }
  res.close();
  //---------------------------------------Tracé des courbes--------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_5.res')\n"
    << "t = A[:,0]\n"
    << "x = A[:,1]\n"
    << "y = A[:,2]\n"
    << "plot(t, x, '+')\n"
    << "plot(t, cos(t))\n"
    << "plot(t, y, '+')\n"
    << "plot(t, -sin(t))\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

ce qui donne le résultat :

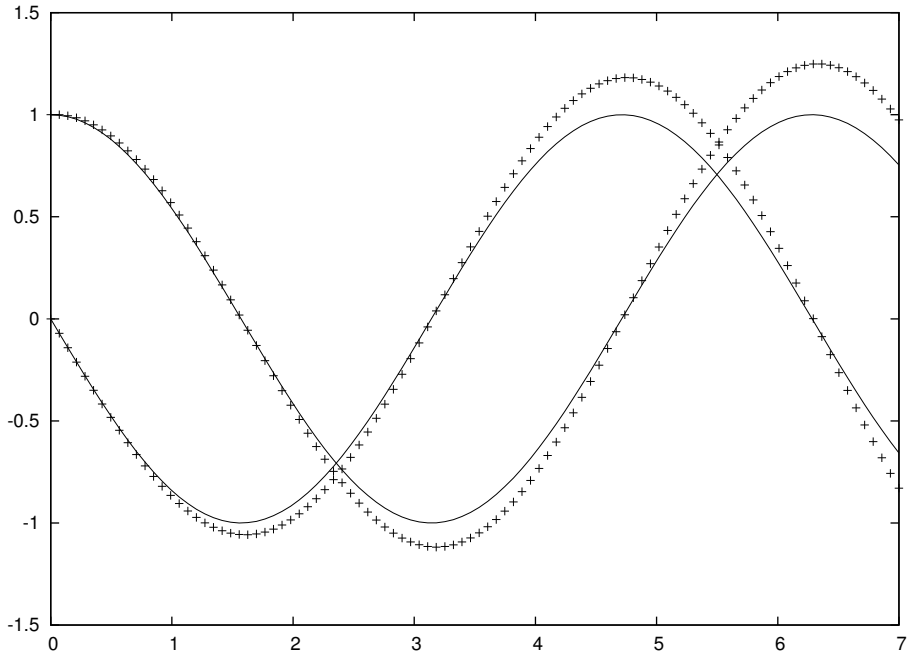


FIGURE 31 –

Les croix représentent le calcul d'Euler, les courbes continues la solution exacte.

Remarques

1. La fonction `euler` reçoit les valeurs des `q` à l'instant `t` ainsi que les valeurs de `t`, `dt` et `n`. Elle appelle alors la fonction `sd` qui calcule et renvoit les `qp` à `euler`. À l'aide de ces `qp`, `euler` calcule les valeurs des `q` à l'instant `t+dt` et les renvoit au programme principal. Finalement l'appel de la fonction `euler` remplace les valeurs des `q` à l'instant `t` par celles à l'instant `t+dt`.
2. Comme dans le cas d'une seule équation, les `qp` ne figurent pas dans le programmme principal, ils sont déclarés par une allocation dynamique dans la fonction `euler`.
3. Dans le cas de cet exemple `n` n'est pas utilisé par la fonction `sd` et il semble inutile de lui transmettre sa valeur. Mais dans le cas où le système différentiel comporte beaucoup d'équations on est amené à mettre des boucles utilisant `n` pour le calcul des `qp`. On le fait donc figurer ici dans un souci de généralité.
4. Toujours dans le cas de cet exemple il ne sert à rien de mettre le temps en argument comme cela est fait car il n'est utilisé ni dans `euler` ni dans `sd`. Mais il y aura des cas où il interviendra dans `sd` comme par exemple dans le système différentiel correspondant à une particule soumise à une force dépendant explicitement du temps et il faudra donc le transmettre à `sd` par l'intermédiaire d'`euler`. On l'introduit donc quand même pour que le programme soit le plus général possible.

3 Système d'équations différentielles du second ordre

Remarque

> Ne pas confondre l'ordre d'un système différentiel, qui est l'ordre de dérivation le plus élevé qui y figure, avec le nombre d'équations qu'il comprend.

Les systèmes d'équations différentielles que nous avons étudiés sont d'ordre un puisqu'ils ne contiennent que les dérivées premières. La méthode se généralise à des systèmes d'ordre supérieur. Considérons par exemple le système de n équations à n fonctions inconnues :

$$\ddot{\vec{r}} = \vec{g}(\vec{r}, \dot{\vec{r}}, t)$$

en notant maintenant $\overrightarrow{r}$ [56] au lieu de $\overrightarrow{q}$ les fonctions inconnues et $\overrightarrow{g}$ au lieu de $\overrightarrow{f}$ le système différentiel.
On peut écrire, comme dans le cas du premier ordre :

$$\overrightarrow{r}(t+dt) = \overrightarrow{r}(t) + dt\ \dot{\overrightarrow{r}}(t)$$

et, de plus :

$$\dot{\overrightarrow{r}}(t+dt) = \dot{\overrightarrow{r}}(t) + dt\ \ddot{\overrightarrow{r}}(t) = \dot{\overrightarrow{r}}(t) + dt\ \overrightarrow{g}\left[\overrightarrow{r}(t), \dot{\overrightarrow{r}}(t), t\right]$$

Donc si on connaît $\overrightarrow{r}(t)$ et $\dot{\overrightarrow{r}}(t)$ on peut calculer $\overrightarrow{r}(t+dt)$ et $\dot{\overrightarrow{r}}(t+dt)$.
Introduisons un vecteur $\overrightarrow{q}$ à 2n composantes définies comme suit :

$$\begin{array}{lcl} q_0 & = & r_0 \\ q_1 & = & r_1 \\ ... & = & ... \\ q_{n-1} & = & r_{n-1} \\ q_n & = & \dot{r}_0 \\ q_{n+1} & = & \dot{r}_1 \\ ... & = & ... \\ q_{2n-1} & = & \dot{r}_{n-1} \end{array}$$

c'est à dire :

$$\begin{array}{lcll} q_i & = & r_i & \text{pour } i = 0...(n-1) \\ q_i & = & \dot{r}_{i-n} & \text{pour } i = n...(2n-1) \end{array}$$

On peut alors écrire :

$$\begin{array}{lcl} \dot{q}_0 & = & q_n \\ \dot{q}_1 & = & q_{n+1} \\ ... & = & ... \\ \dot{q}_i & = & q_{n+i} \\ ... & = & ... \\ \dot{q}_{n-1} & = & q_{2n-1} \\ \dot{q}_n & = & g_0(\overrightarrow{q}, t) \\ \dot{q}_{n+1} & = & g_1(\overrightarrow{q}, t) \\ ... & = & ... \\ \dot{q}_i & = & g_{i-n}(\overrightarrow{q}, t) \\ ... & = & ... \\ \dot{q}_{2n-1} & = & g_{n-1}(\overrightarrow{q}, t) \end{array}$$

Si on introduit une fonction $\overrightarrow{f}$ à $2n$ composantes définies comme suit :

$$\begin{array}{lcl} f_0(\overrightarrow{q}, t) & = & q_n \\ f_1(\overrightarrow{q}, t) & = & q_{n+1} \\ ... & = & ... \\ f_{n-1}(\overrightarrow{q}, t) & = & q_{2n-1} \\ f_n(\overrightarrow{q}, t) & = & g_0(\overrightarrow{q}, t) \\ f_{n+1}(\overrightarrow{q}, t) & = & g_1(\overrightarrow{q}, t) \\ ... & = & ... \\ f_{2n-1}(\overrightarrow{q}, t) & = & g_{n-1}(\overrightarrow{q}, t) \end{array}$$

c'est à dire :

$$\begin{array}{lcll} f_i(\overrightarrow{q}, t) & = & q_{n+i} & \text{pour } i = 0...(n-1) \\ f_i(\overrightarrow{q}, t) & = & g_{i-n}(\overrightarrow{q}, t) & \text{pour } i = n...(2n-1) \end{array}$$

56. Cette notation $\overrightarrow{r}$ n'a évidemment en général rien à voir avec la notation $\overrightarrow{r} = \overrightarrow{OM}$ du rayon-vecteur d'un point dans l'espace à deux ou trois dimensions.

On peut alors écrire :

$$\begin{array}{lcl}
\dot{q}_0 & = & f_0(\overrightarrow{q},t) \\
\dot{q}_1 & = & f_1(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_i & = & f_i(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_{n-1} & = & f_{n-1}(\overrightarrow{q},t) \\
\dot{q}_n & = & f_n(\overrightarrow{q},t) \\
\dot{q}_{n+1} & = & f_{n+1}(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_i & = & f_i(\overrightarrow{q},t) \\
... & = & ... \\
\dot{q}_{2n-1} & = & f_{2n-1}(\overrightarrow{q},t)
\end{array}$$

soit :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q},t)$$

et le système d'ordre deux (de dimension n) est ramené à un système d'ordre un (de dimension $2n$). On peut généraliser à un système d'ordre quelconque pourvu que les dérivées d'ordre le plus élevé soient exprimées explicitement en fonction de celles d'ordre moins élevé.
Exemple :

$$\begin{array}{lcl}
\ddot{r}_0 & = & r_1 - \dfrac{t}{4}\dot{r}_1 + 6t \\
\ddot{r}_1 & = & r_0 + 4\dot{r}_0 - \dfrac{\dot{r}_1}{4}
\end{array}$$

On pose :

$$\begin{array}{lcl}
q_0 & = & r_0 \\
q_1 & = & r_1 \\
q_2 & = & \dot{r}_0 \\
q_3 & = & \dot{r}_1
\end{array}$$

et le système différentiel s'écrit :

$$\begin{array}{lcl}
\dot{q}_0 & = & q_2 \\
\dot{q}_1 & = & q_3 \\
\dot{q}_2 & = & q_1 - \dfrac{t}{4}q_3 + 6t \\
\dot{q}_3 & = & q_0 + 4q_2 - \dfrac{q_3}{4}
\end{array}$$

Puisqu'on s'est ramené à la résolution d'un système du premier ordre la programmation du calcul pas à pas des solutions entre dans le cadre du programme `euler_5.cpp` précédemment écrit dans lequel il suffit de changer :

— l'expression du système différentiel dans la fonction `sda`
— le nombre d'équations et les conditions initiales dans le `main`.

Ce qui donne :

```
// euler_6.cpp
// On intègre un système d'équations différentielles du second ordre ramené au premier ordre
#include<bibli_fonctions.h>
//-----------------------Fonction système d'équations différentielles-------------------------
void sda(double* q, double t, double* qp, int n) {
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = q[1] - t*q[3]/4 + 6*t;
  qp[3] = q[0] + 4*q[2] - q[3]/4;
}
//--------------------------------------Fonction euler-----------------------------------------
void euler(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i;
  double* qp = (double*)malloc(n*sizeof(double)); // ou bien double* qp = D_1(n) si on utilise
  sd(q,t,qp,n);                                   // les fonctions du Magistère
  for(i = 0; i < n; i++)
```

```
    q[i] = q[i] + dt*qp[i];                          // calcul d'Euler
  free(qp);   // très important ici parce qu'Euler est appelée un très grand nombre de fois
              // (ou bien f_D_1(qp,n) si on utilise des fonctions du Magistère)
}
//----------------------------------------Main-----------------------------------------------
int main() {
  int i, np, n = 4;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double)); // ou bien double* q = D_1(n) si on utilise
  fstream res;                                   // les fonctions du Magistère
  res.open("euler_6.res", ios::out);
  np = 100; tfin = 1.5; dt = tfin/(np-1);        // valeurs des paramètres
  t = 0; q[0] = 0; q[1] = 0; q[2] = 0; q[3] = 0; // conditions initiales
  for(i = 1; i <= np; i++) {                     // boucle sur t
    res << t << " " << q[0] << " " << q[1] << endl;
    euler(sda,q,t,dt,n);
    //rk4(sda,q,t,dt,n);  // seule ligne à changer si on fait Runge-Kutta au lieu d'Euler
    t = t+dt;             // (voir ci-dessous la présentation de la méthode de Runge-Kutta)
  }
  res.close();
  //----------------------------------------Tracé des courbes---------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('euler_6.res')\n"
    << "t = A[:,0]\n"
    << "x = A[:,1]\n"
    << "y = A[:,2]\n"
    << "plot(t, x, '+')\n"
    << "plot(t, t**3)\n"
    << "plot(t, y, '+')\n"
    << "plot(t, t**4)\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

avec le résultat :

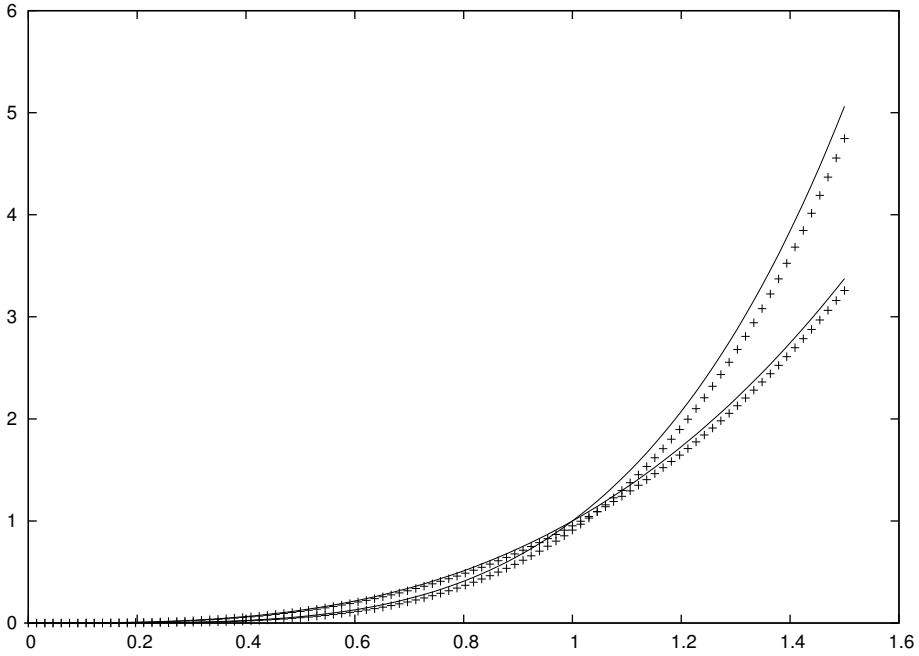


FIGURE 32 –

Les croix représentent le calcul d'Euler, les courbes continues la solution exacte ($r_0 = t^3, r_1 = t^4$).

Remarques

1. Le programme `euler_5.cpp` (ou aussi bien `euler_6.cpp`) permet ainsi en principe de résoudre un système d'un nombre quelconque d'équations et d'ordre quelconque, dès lors que les équations sont résolues par rapport aux dérivées d'ordre le plus élevé. Il suffit d'écrire le système dans la fonction système différentiel, notée ici `sda`, et de donner les conditions initiales avant la boucle sur la variable `t`.
2. Jusqu'ici on ne s'est pas posé la question de savoir comment choisir la valeur du pas dt. S'il est trop grand l'approximation pas à pas est trop imprécise, s'il est trop petit le calcul est inutilement long ou faussé par les erreurs de troncature. Il faut choisir un pas d'essai en se guidant sur les propriétés mathématiques (ou physiques s'il s'agit d'un problème de physique) du système d'équations étudié et faire un premier calcul avec ce pas. Ensuite on fait un second calcul avec un pas deux fois plus petit et on compare les résultats des deux calculs. S'ils diffèrent notablement on continue à diviser le pas par deux jusqu'à ce que les résultats se stabilisent. Cette procédure peut éventuellement être exécutée par le programme lui même.
   En pratique dans le programme `euler_5.cpp` ce n'est pas directement le pas dt qu'on choisit mais l'intervalle $[t_{\text{initial}}, t_{\text{final}}]$ sur lequel on veut intégrer le système d'équations différentielles et le nombre n_p de points que l'on veut calculer. Le pas en découle par la formule $dt = \dfrac{t_{\text{final}} - t_{\text{initial}}}{n_p - 1}$.

4 Méthode de Runge-Kutta d'ordre 4

C'est, comme la méthode d'Euler, une méthode pas à pas permettant de calculer une valeur approchée de $q(t+dt)$ à partir de celle de $q(t)$. La formule d'itération est plus compliquée que celle de la méthode d'Euler mais l'approximation est meilleure pour une même valeur du pas. Elle s'applique aux systèmes de même type que ceux étudiés par la méthode d'Euler :

$$\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q}, t)$$

Les formules d'itération sont les suivantes :

$$\overrightarrow{q}(t+dt) = \overrightarrow{q}(t) + \frac{dt}{6}(\overrightarrow{p}_1 + 2\,\overrightarrow{p}_2 + 2\,\overrightarrow{p}_3 + \overrightarrow{p}_4)$$

avec :

$$\overrightarrow{p}_1 = \overrightarrow{f}(t, \overrightarrow{q}(t))$$

$$\overrightarrow{p}_2 = \overrightarrow{f}(t + \frac{dt}{2}, \overrightarrow{q}(t) + \frac{dt}{2}\overrightarrow{p_1})$$

$$\overrightarrow{p}_3 = \overrightarrow{f}(t + \frac{dt}{2}, \overrightarrow{q}(t) + \frac{dt}{2}\overrightarrow{p_2})$$

$$\overrightarrow{p}_4 = \overrightarrow{f}(t + dt, \overrightarrow{q}(t) + dt\,\overrightarrow{p_3})$$

Il y a donc quatre quantités intermédiaires à calculer ($\overrightarrow{p}_1$, $\overrightarrow{p}_2$, $\overrightarrow{p}_3$, $\overrightarrow{p}_4$) pour obtenir $\overrightarrow{q}(t+dt)$.
La fonction C, qui calcule $\overrightarrow{q}(t+dt)$ en fonction de $\overrightarrow{q}(t)$ à l'aide de ces formules est donnée en annexe. Elle est nommée `rk4(f,q,t,dt,n)` et s'emploie exactement comme `euler(f,q,t,dt,n)`. A chaque appel, les valeurs de $\overrightarrow{q}(t)$, contenues dans le tableau `q`, sont remplacées par les valeurs de $\overrightarrow{q}(t+dt)$, `f` étant la fonction qui définit le système différentiel.

Considérons à nouveau le système :

$$\begin{aligned} \dot{q}_0 &= q_1 \\ \dot{q}_1 &= -q_0 \end{aligned}$$

précédemment étudié avec la méthode d'Euler. Pour faire le calcul avec la méthode Runge-Kutta d'ordre 4 il suffit de remplacer l'appel `euler(ed1,q,t,dt,n);` par `rk4(ed1,q,t,dt,n);`[57] dans la boucle sur t du programme précédent `euler_5.cpp`. On obtient le le résultat suivant :

57. La fonction `rk4` fait partie de la bibliothèque du Magistère

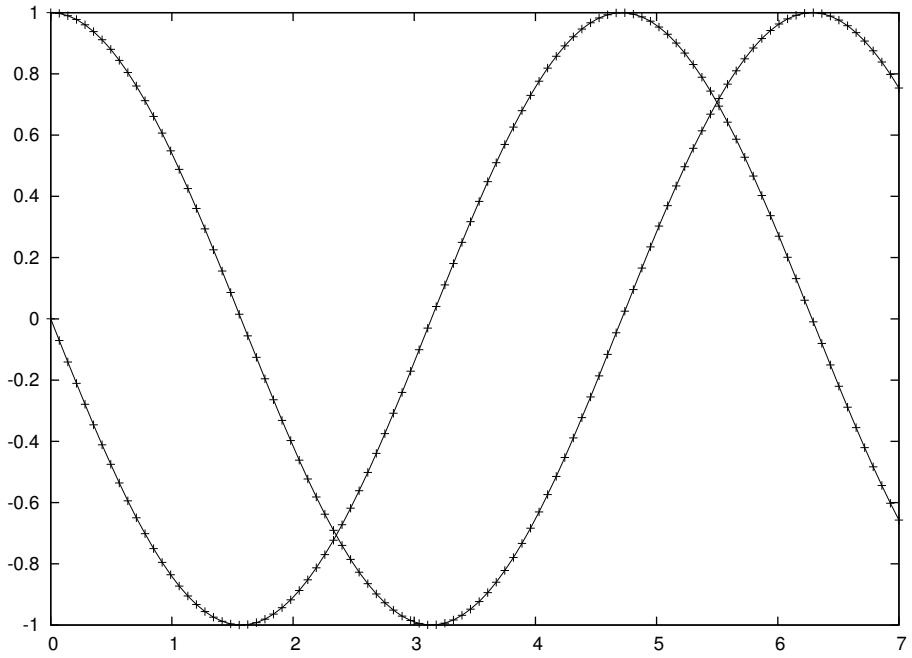


FIGURE 33 –

Les croix représentent le calcul RK4, les pointillés la solution exacte. Les croix suivent les pointillés à la précision de la figure : on vérifie ainsi que, pour un pas identique, la précision du calcul RK4 est meilleure que celle du calcul d'Euler.

Dans la suite on va montrer l'application de la méthode de Runge-Kutta d'ordre 4 à un certain nombre d'exemples pris en physique. On emploiera systématiquement la méthode RK4, bien plus précise que celle d'Euler, cette dernière ayant été utilisée au début du chapitre uniquement parce qu'elle permet d'expliquer simplement le principe des méthodes pas à pas et de leur programmation.
La méthode d'ajustement du pas est la même pour RK4 que pour Euler.

5 Exemples d'application de la méthode de Runge-Kutta d'ordre 4

5.1 Exemple : régime transitoire et régime permanent d'un oscillateur

Une masse m liée sans frottement à une tige rectiligne est rappelée par un ressort de raideur k. Elle est soumise de plus :

à une force de frottement fluide proportionnelle à la vitesse
à une force extérieure sinusoïdale $f \sin \omega t$.

Sa position $q(t)$ obéit à l'équation différentielle du second ordre :

$$\ddot{q} = -\frac{l}{m}\dot{q} - \frac{k}{m}q + \frac{f}{m}\sin \omega t$$

On pose :

$$q_0 = q$$
$$q_1 = \dot{q}$$

et :

$$a = \frac{l}{m}$$
$$\omega_0 = \sqrt{\frac{k}{m}}$$

L'équation différentielle du second ordre est équivalente au système du premier ordre :

$$\dot{q}_0 = q_1$$
$$\dot{q}_1 = -aq_1 - \omega_0^2 q_0 + \frac{f}{m}\sin\omega t$$

qui peut être résolu numériquement en utilisant la fonction `rk4` dans le programme suivant :

```
// oscill.cpp
#include<bibli_fonctions.h>
void osc(double* q, double t, double* qp, int n) {
  static double l = 0.1, om0 = 3.7, f = 2., om = 1.;
  qp[0] = q[1];
  qp[1] = -l*q[1] - om0*om0*q[0] + f*sin(om*t);
}
int main() {
  int i, np, n = 2;
  double t, dt, tfin;
  double* q = (double*)malloc(n*sizeof(double));
  fstream res;
  res.open("oscill.res", ios::out);
  t = 0.; q[0] = 1.; q[1] = 0.;
  np = 2000; tfin = 120.; dt = tfin/(np-1);
  for(i = 1; i <= np; i++) {
    res << t << " " << q[0] << " " << q[1] << endl;
    rk4(osc,q,t,dt,n);
    t = t+dt;
  }
  res.close();
  //---------------------------------------Tracé des courbes--------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('oscill.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

On obtient la courbe suivante pour $q(t)$:

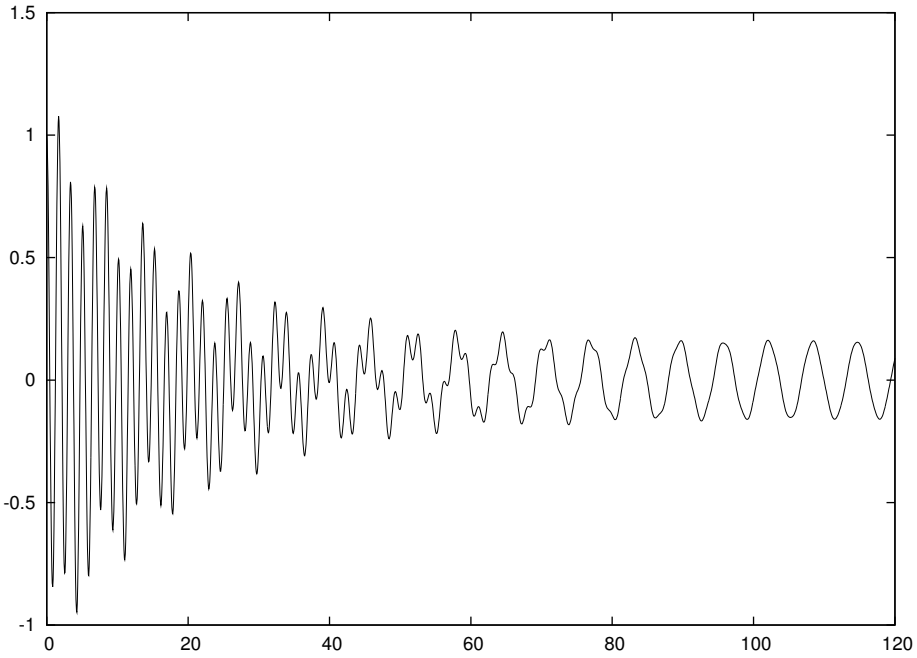


FIGURE 34 –

montrant le passage progressif des oscillations libres amorties aux oscillations entretenues et en particulier le changement de fréquence.

5.2 Pendule à deux degrés de liberté

Un pendule est constitué d'une tige rigide de longueur l et de masse négligeable attachée à un point fixe O et dont l'autre extrémité porte une masse ponctuelle m. La tige tourne librement autour du point O. On lance le pendule depuis une position quelconque avec une vitesse quelconque, ce qui fait qu'en général le mouvement ne se fait pas dans un plan. On ne fait pas l'approximation des petites oscillations.

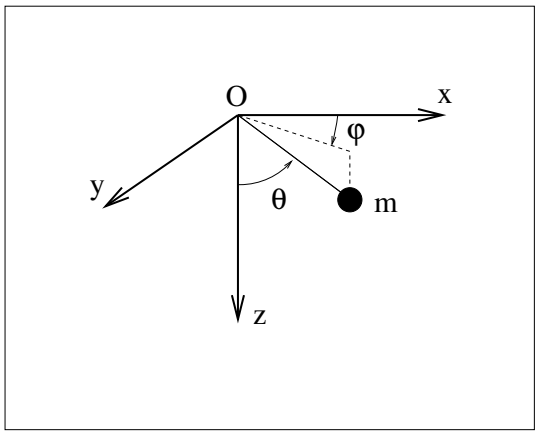


FIGURE 35 –

Dans ces conditions les angles θ et φ qui repèrent la position du pendule obéissent au système d'équations différentielles :

$$\ddot{\theta} = \frac{1}{2}\sin 2\theta\, \dot{\varphi}^2 - \frac{g}{l}\sin\theta$$
$$\ddot{\varphi} = -\frac{2\,\dot{\theta}\,\dot{\varphi}}{\tan\theta}$$

Le programme peut s'écrire :

```
// pendule_2_degres_liberte.cpp
#include<bibli_fonctions.h>
double g = 9.81, l;
void el(double* q, double t, double* qp, int n) {
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = sin(2*q[0])*q[3]*q[3]/2 - g/l*sin(q[0]);
  qp[3] = -2/tan(q[0])*q[2]*q[3];
}
int main() {
  int i, np, n = 4;
  double t,dt,tfin;
  double* q = (double*)malloc(n*sizeof(double));
  fstream res;
  res.open("pendule_2_degres_liberte.res", ios::out);
  // Unités : SI
  l = g/2/M_PI/2/M_PI;
  // Conditions initiales
  t = 0; q[0] = M_PI/2; q[1] = 0; q[2] = 0; q[3] = 0.2;
  np = 100000; tfin = 20; dt = tfin/(np-1);
  // Boucle sur le temps
  for(i = 1; i <= np; i++) {
    res << l*sin(q[0])*cos(q[1]) << " " << l*sin(q[0])*sin(q[1]) << endl;
    rk4(el,q,t,dt,n);
    t += dt;
  }
  res.close();
  //---------------------------------------Tracé des courbes---------------------------------
  ostringstream pyth;
  pyth
    << "A = loadtxt('pendule_2_degres_liberte.res')\n"
    << "plot(A[:,0], A[:,1])\n"
    ;
  make_plot_py(pyth);
  return 0;
}
```

et on obtient la courbe suivante pour la projection de la position de la masse dans le plan xOy :

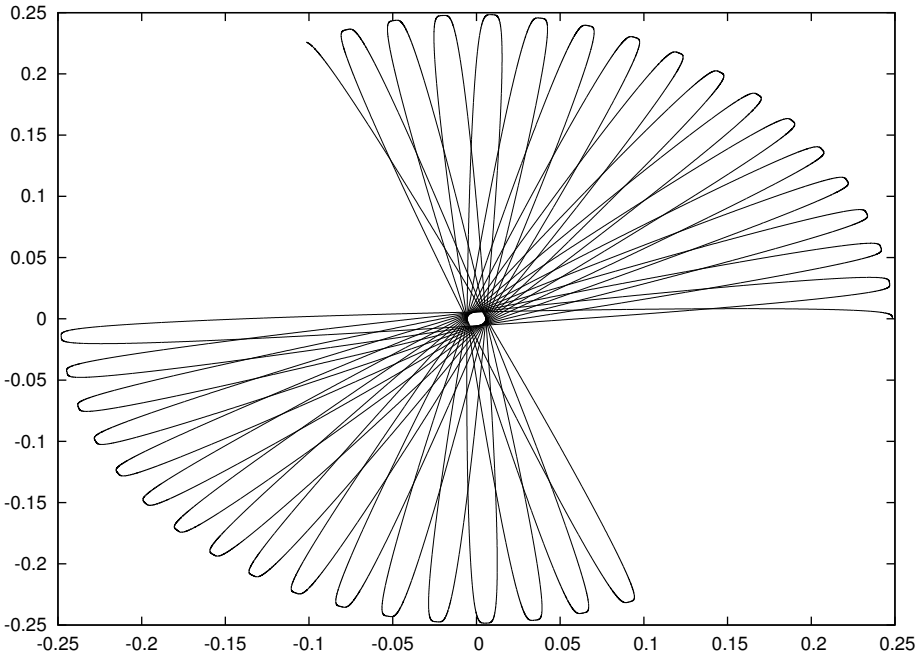


FIGURE 36 –

5.3 Mouvement d'un satellite en présence de frottement

Un satellite est soumis à l'attraction de la terre et à une force de frottement fluide proportionnelle au carré de sa vitesse.

$$\ddot{\overrightarrow{r}} = -gr_t^2 \frac{\overrightarrow{r}}{r^3} - \frac{k}{m}\dot{r}\dot{\overrightarrow{r}}$$

avec :

- $\overrightarrow{r}$ vecteur position du satellite, de composantes x et y
- g accélération de la pesanteur au niveau du sol
- r_t rayon de la terre
- m masse du satellite

C'est un système de deux équations du second ordre que l'on peut ramener à quatre équations du premier ordre. On pose :

$$a = gr_t^2$$
$$b = \frac{k}{m}$$

et :

$$q_0 = x$$
$$q_1 = y$$
$$q_2 = \dot{x}$$
$$q_3 = \dot{y}$$

Le système s'écrit :

$$\dot{q}_0 = q_2$$
$$\dot{q}_1 = q_3$$
$$\dot{q}_2 = -a\frac{q_0}{(q_0^2+q_1^2)^{\frac{3}{2}}} - b\sqrt{q_2^2+q_3^2}\, q_2$$
$$\dot{q}_3 = -a\frac{q_1}{(q_0^2+q_1^2)^{\frac{3}{2}}} - b\sqrt{q_2^2+q_3^2}\, q_3$$

La fonction définissant l'équation différentielle peut s'écrire :

```
void sa(double* q, double t, double* qp, int n) {
  double r3, v;
  r3 = pow(q[0]*q[0]+q[1]*q[1], 3./2.);
  v = sqrt(q[2]*q[2]+q[3]*q[3]);
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = -a*q[0]/r3 - b*v*q[2];
  qp[3] = -a*q[1]/r3 - b*v*q[3];
}
```

et on obtient, comme attendu, des trajectoires du type :

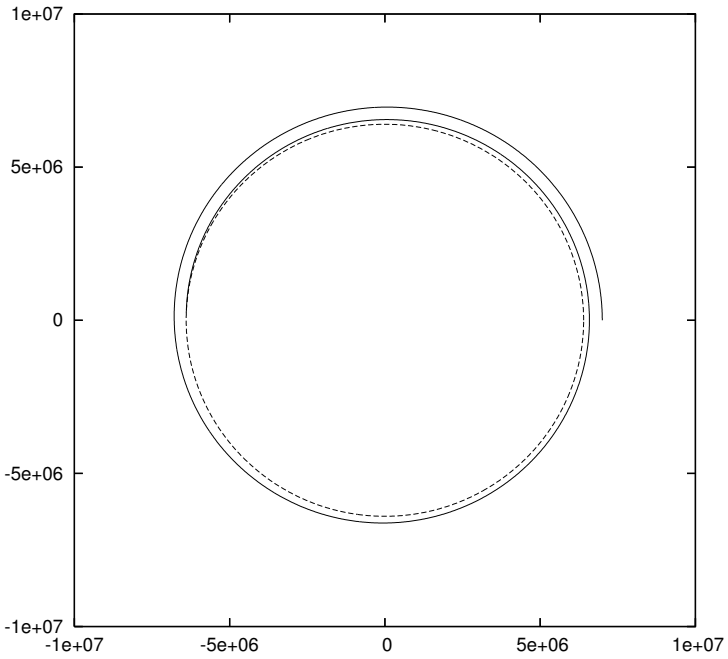


FIGURE 37 – La surface de la terre est représentée en pointillés

Remarque

On n'a pas tenu compte de l'augmentation de la densité de l'air et donc du frottement quand l'altitude diminue. Il serait très facile de le faire en remplaçant le coefficient `b` par une fonction de l'altitude.

5.4 Électron soumis à l'attraction d'un noyau fixe et à un champ homogène

Les équations sont voisines de celles du satellite :

$$\ddot{\overrightarrow{r}} = -\frac{ze^2}{4\pi\varepsilon_0 m}\frac{\overrightarrow{r}}{r^3} + \frac{e}{m}\overrightarrow{E}$$

e est la charge algébrique de l'électron, ze celle de la charge fixe, $\overrightarrow{E}$ le champ homogène.
On pose :

$$a = -\frac{ze^2}{4\pi\varepsilon_0 m}$$

$$b_x = \frac{e}{m}E_x$$

$$b_y = \frac{e}{m}E_y$$

et, comme pour le satellite :

$$\begin{aligned} q_0 &= x \\ q_1 &= y \\ q_2 &= \dot{x} \\ q_3 &= \dot{y} \end{aligned}$$

Le système s'écrit :

$$\dot{q}_0 = q_2$$

$$\dot{q}_1 = q_3$$

$$\dot{q}_2 = a\frac{q_0}{(q_0^2+q_1^2)^{\frac{3}{2}}} + b_x$$

$$\dot{q}_3 = a\frac{q_1}{(q_0^2+q_1^2)^{\frac{3}{2}}} + b_y$$

la fonction qui définit le système différentiel s'écrit :

```
void el(double* q, double t, double* qp, int n) {
  double r3;
  r3 = pow(q[0]*q[0]+q[1]*q[1], 3./2.);
  qp[0] = q[2];
  qp[1] = q[3];
  qp[2] = a*q[0]/r3 + bx;
  qp[3] = a*q[1]/r3 + by;
}
```

On obtient des courbes qui ont une grande variété de formes en fonction des conditions initiales et de $\overrightarrow{E}$. Un exemple :

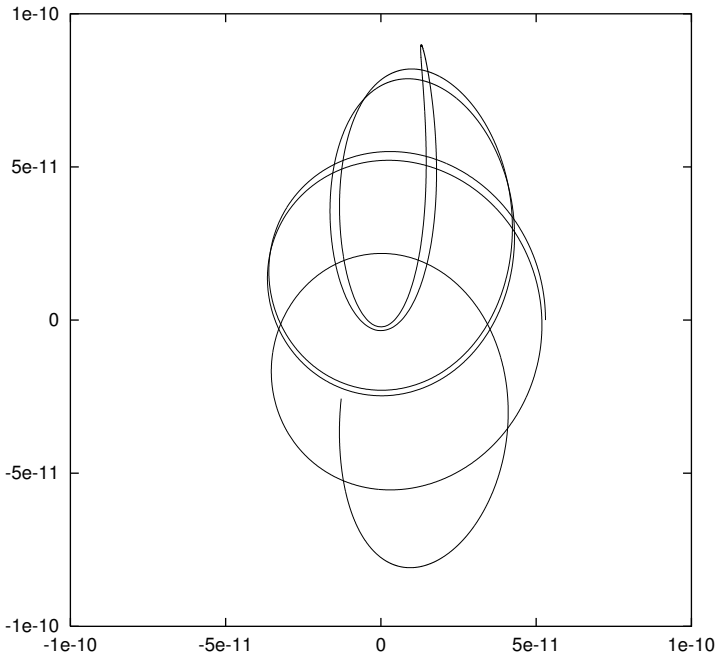


FIGURE 38 –

Remarque

Bien entendu ce système relève de la mécanique quantique (effet Stark) et non de la mécanique classique. Mais il est souvent intéressant d'étudier si des propriétés d'un système classique se retrouvent dans le système quantique correspondant.
D'autre part, les équations précédentes et les trajectoires en résultant sont exactement celles d'un satellite

placé dans le champ gravitationnel de la terre et soumis à une pression de radiation comme celle du rayonnement solaire (voile solaire).

6 Conclusion

Les formules de Runge-Kutta d'ordre 4 sont un outil adapté à l'étude de nombreux problèmes de physique. Mais :

- il faudrait ajouter un ajustement automatique du pas d'itération : s'il reste fixe il est en général inutilement petit pour certaines parties du calcul (perte de temps) et trop grand pour d'autres (perte de précision)
- même avec un ajustement du pas, l'erreur due aux approximations peut augmenter très rapidement au fur et à mesure des itérations. Dans le cas de systèmes instables, cela peut interdire tout calcul. Le résultat doit toujours être examiné d'un œil critique à défaut d'une vraie étude numérique garantissant une certaine précision. Par exemple il faut vérifier, quand c'est possible, si certains invariants du système sont respectés.

Par ailleurs il existe d'autres méthodes de résolution numérique des équations différentielles, plus efficaces pour certains problèmes.
Enfin, un autre domaine très important pour la physique est celui des équations aux dérivées partielles. Il n'est pas abordé ici et exige des techniques nettement plus compliquées.

Annexe : programmation des formules de Runge-Kutta d'ordre 4, fonction `rk4`

```
#include<bibli_fonctions.h>
void rk4(void(*sd)(double*,double,double*,int), double* q, double t, double dt, int n) {
  int i, k, p, PM = 4;
  static const double c2 = 1./2, c3 = 1./3, c6 = 1./6;
  /* Allocations et initialisations */
  double** a = (double**)malloc(PM*sizeof(double*));
  for(i = 0; i < PM; i++)
    a[i] = (double*)malloc(PM*sizeof(double));
  ini_D_2(a,PM,PM,c2,0.,0.,0.,0.,c2,0.,0.,0.,0.,1.,0.,c6,c3,c3,c6);
  double* b = (double*)malloc(PM*sizeof(double));
  ini_D_1(b,PM,0.,c2,c2,1.);
  double** y = (double**)malloc((PM+1)*sizeof(double*));
  for(i = 0; i < PM+1; i++)
    y[i] = (double*)malloc(n*sizeof(double));
  double** z = (double**)malloc(PM*sizeof(double*));
  for(i = 0; i < PM; i++)
    z[i] = (double*)malloc(n*sizeof(double));
  /* Calcul */
  for(i = 0; i < n; i++)
    y[0][i] = q[i];
  for(p = 1; p <= PM; p++) {
    sd(y[p-1], t+b[p-1]*dt, z[p-1], n);
    for(i = 0; i < n; i++)
      y[p][i] = q[i];
    for(k = 0; k < p; k++)
      for(i = 0; i < n; i++)
        y[p][i] = y[p][i] + dt*a[p-1][k]*z[k][i];
  }
  for(i = 0; i < n; i++)
    q[i] = y[PM][i];
  /* Desallocations */
  f_D_2(a,PM,PM); f_D_1(b,PM); f_D_2(y,PM+1,n); f_D_2(z,PM,n);
}
```

Remarque

Il ne faut pas recopier cette fonction. Elle est disponible en mettant la directive :

```
#include<bibli_fonctions.h>
```

en tête du programme qui l'utilise et en compilant et exécutant par *ccc*.

Annexe : exemple de trois masses en interaction gravitationnelle

On considère trois particules ponctuelles de masses quelconques, s'attirant selon la loi de la gravitation universelle. On se place dans un repère galiléen quelconque, d'origine O. La relation fondamentale de la dynamique appliquée à chacune des particules s'écrit :

$$m_\alpha \frac{d^2\overrightarrow{OM_\alpha}}{dt^2} = Gm_\alpha \sum_{\beta\neq\alpha} \frac{m_\beta \overrightarrow{M_\alpha M_\beta}}{(M_\alpha M_\beta)^3} \qquad \text{avec} \quad \alpha = 0,1,2 \quad \beta = 0,1,2$$

$M_\alpha M_\beta$ est la distance des particules M_α et M_β, G la constante de la gravitation universelle $6.67\ 10^{-11}$ MKSA. On note $x_{i\alpha}$ la $i^{\text{ème}}$ coordonnée cartésienne de la $\alpha^{\text{ème}}$ particule. On a alors :

$$\ddot{x}_{i\alpha} = G \sum_{\beta\neq\alpha} \frac{m_\beta (x_{i\beta} - x_{i\alpha})}{(M_\alpha M_\beta)^3} = f_{i\alpha}(x_{00}, x_{10}, \ldots, x_{22}) \qquad \text{avec} \quad i = 0,1,2 \qquad \text{(A)}$$

Posons :

$$\begin{aligned} q_0 &= x_{00} \\ q_1 &= x_{10} \\ \cdots &= \cdots \\ q_3 &= x_{01} \\ q_4 &= x_{11} \\ \cdots &= \cdots \\ q_8 &= x_{22} \end{aligned}$$

et :

$$\begin{aligned} q_9 &= \dot{q}_0 \\ q_{10} &= \dot{q}_1 \\ \cdots &= \cdots \\ q_{17} &= \dot{q}_8 \end{aligned}$$

Les équations (A) peuvent alors s'écrire :

$$\begin{aligned} \dot{q}_0 &= q_9 \\ \dot{q}_1 &= q_{10} \\ \cdots &= \cdots \\ \dot{q}_8 &= q_{17} \\ \dot{q}_9 &= f_{00}(q_0 \ldots q_8) \\ \dot{q}_{10} &= f_{10}(q_0 \ldots q_8) \\ \cdots &= \cdots \\ \dot{q}_{17} &= f_{22}(q_0 \ldots q_8) \end{aligned}$$

On est donc ramené à un système de la forme $\dot{\overrightarrow{q}} = \overrightarrow{f}(\overrightarrow{q})$ avec un vecteur $\overrightarrow{q}$ à 18 composantes[58], dont les solutions peuvent être calculées en utilisant la fonction `rk4`, compte-tenu des réserves énoncées dans la conclusion du chapitre.

58. Dans ce cas particulier la fonction $\overrightarrow{f}$ ne dépend pas explicitement du temps.

12. Structures

On présente ici quelques éléments de base pour définir des structures simples en C. Pour des cas plus complexes il vaut mieux utiliser les classes du C++.

1 Définition

Une structure est un ensemble d'éléments de types variés tels que constantes, variables, pointeurs, regroupés sous un même nom. Ce nom, choisi par l'utilisateur, permet, au choix, de traiter l'ensemble comme un seul bloc, ou chaque élément séparément. Pour les structures les éléments sont appelés « membres » ou « champs » . Ici on utilisera le terme « membres » .

2 Déclaration

2.1 Déclaration de base, membres

On déclare le type de la structure de la façon suivante, par exemple :

```
struct stru {
  int i;
  double x, y;
  char c;
};
```

`struct` est un mot clé du C, `stru` le nom choisi par l'utilisateur pour le type de la structure, `i`, `x`, `y` et `c` les noms des membres choisis par l'utilisateur. Dans cet exemple la structure comprend un `int`, deux `double` et un `char`, on pourrait évidemment en mettre plus, ou ajouter des pointeurs de tous types.
`i`, `x`, `y` et `c` ne sont pas des variables, ce sont des noms de membres.
On déclare ensuite une structure du type `stru` par :

```
struct stru ss;
```

`ss` étant le nom de la structure choisi par l'utilisateur. On peut alors désigner la structure dans son ensemble par son nom, ou ses membres séparément avec l'opérateur « . » : `ss.i` est la variable de type `int` contenue dans la structure `ss`, `ss.x` est la première variable de type `double` contenue dans la structure `ss`, etc..

2.2 Déclaration simplifiée avec `typedef`

Si on ajoute à la déclaration de structure précédente :

```
typedef struct stru struty;
```

alors toutes les déclarations telles que :

```
struct stru ss;
```

peuvent être simplifiées en :

```
struty ss;
```

Comme d'habitude, le nom du type (ici `struty`) peut être choisi librement par l'utilisateur. Par contre, `typedef` est un mot clé du C.
Comme pour les variables ordinaires on peut déclarer plusieurs structures du même type simultanément :

```
struty ss1, ss2, ss3;
```

On peut combiner la déclaration du type de la structure et le `typedef` en une seule commande (ce qui est en fait la façon préférée de déclarer un type de structure) :

```
typedef struct {
  int i;
  double x, y;
  char c;
} struty;
```

On voit que dans ce cas il n'est pas nécessaire d'introduire le nom `stru` intermédiaire.

2.3 Règles

Un membre d'un type structure peut être d'un type structure défini par ailleurs, mais pas du type structure auquel il appartient.
Si un type de structure est déclaré dans une fonction (`main` compris), ce type n'est accessible que dans cette fonction. S'il est déclaré en dehors de toute fonction il est accessible depuis toute fonction située après cette déclaration (même règle que pour l'accessibilité d'une variable).
Contrairement au cas des variables il n'est pas possible d'accéder à un type de structure déclaré dans un fichier différent.

3 Utilisation

3.1 Initialisation

Exemple d'initialisation :

```
#include<iostream>
using namespace std;
int main() {
  typedef struct {
    int i;
    double x, y;
    char c;
  } struty;
  struty ss = {5, 1.26, 2.34, 'K'};
  cout << ss.i << " " << ss.x << " " << ss.y << " " << ss.c << endl;
  return 0;
}
```

3.2 Affectation : utilisation du signe =

On peut affecter la valeur d'une structure à une autre structure avec le signe = comme pour des variables simples en écrivant :

```
ss2 = ss1;
```

à condition, bien entendu, que `ss1` ait été initialisée auparavant et que les deux structures `ss1` et `ss2` soient du même type (ici `struty`). La valeur de chaque membre de `ss1` est alors attribuée au membre correspondant de `ss2`. Les membres étant des variables comme les autres rien n'interdit d'écrire également :

```
ss2.i = ss1.i;
ss2.x = ss1.x;
...
```

pour tout ou partie des membres de `ss1` et `ss2`.
On ne peut, par contre, pas comparer deux structures de même type à l'aide des opérateurs == ou !=, il faut comparer membre à membre.

3.3 Adresse et taille d'une structure

L'adresse s'obtient comme pour une variable simple, en utilisant l'opérateur &.
L'opérateur `sizeof` donne la taille de la structure, qui est égale ou supérieure à la somme de celles de ses membres.

3.4 Passage d'une structure en argument d'une fonction

Comme pour une variable simple on peut passer par valeur ou par adresse.

3.4.1 Passage par valeur

```
... f(..., struty zz, ...) {
  ...
}
int main() {
  ...
  struty ss;
  ...
  f(..., ss, ...);
  ...
}
```

Même si les membres de `zz` sont modifiés ceux de `ss` ne le sont pas.

3.4.2 Passage par adresse

```
... f(..., struty *zz, ...) {
  ...
}
int main() {
  ...
  struty ss;
  ...
  f(..., &ss, ...);
  ...
}
```

Dans la fonction `f`, `zz` n'est plus une structure mais un pointeur sur une structure. Pour accéder à la structure dont l'adresse se trouve dans le pointeur (c'est à dire ici à la structure `ss`), il faut écrire `*zz` et pour accéder aux membres de cette structure `(*zz).i` [59], `(*zz).x`, etc.. Il existe une autre notation pour désigner `(*zz).i` qui est `zz->i`, et qui permet donc d'accéder aux membres d'une structure en utilisant son adresse de début.
Si les membres de `*zz` sont modifiés ceux de `ss` le sont.

3.5 Fonction de type structure

Une fonction peut être de type structure :

```
struty f(...) {
  struty xx;
  ...
  return xx;
}
```

4 Application : dioptre sphérique

On considère un dioptre sphérique dont l'axe optique est orienté dans le sens de propagation de la lumière.

59. Les parenthèses sont indispensables car « . » est prioritaire sur « * » .

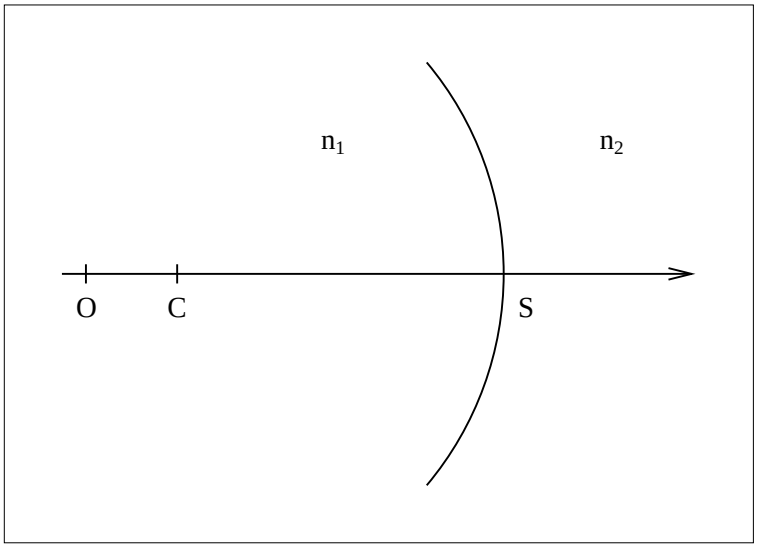


FIGURE 39 –

x est l'abscisse du sommet S du dioptre par rapport à une origine O quelconque de l'axe, et C étant le centre de courbure du dioptre, on pose $r = \overline{SC}$ (rayon de courbure algébrique). On note n_1 l'indice avant le dioptre, n_2 après. Le programme suivant permet de définir un dioptre par ses quatre paramètres x, r, n_1, n_2 et calcule sa matrice de transfert. On pourrait le développer en définissant un ensemble de dioptres dans un tableau de dioptres et en calculant la matrice de transfert du système total, puis faire tracer le trajet des rayons lumineux à travers les dioptres successifs, étudier l'aberration chromatique.

```
// Ce programme calcule la matrice d'un dioptre
#include<iostream>
#include<stdlib.h>
using namespace std;
//-----------------------------------------------------------------------
typedef struct {
  double x, sc, n1, n2;
} diop;
//-----------------------------------------------------------------------
void matrice(diop a, double **mm) {
  mm[0][0] = 1.;
  mm[0][1] = 0.;
  mm[1][0] = (a.n1 - a.n2) / a.n2 / a.sc;
  mm[1][1] = a.n1 / a.n2;
}
//-----------------------------------------------------------------------
int main() {
  const int nn = 2;
  int i, j;
  diop d = {1., -1., 1., 1.5};
  double** mm = (double**)malloc(nn * sizeof(double*));
  for(i = 0; i < nn; i++)
    mm[i] = (double*)malloc(nn * sizeof(double));
  // ou double** mm = D_2(nn,nn); en utilisant les fonctions du Magistère
  matrice(d,mm);
  for(i = 0; i < nn; i++) {
    for(j = 0; j < nn; j++)
      cout << mm[i][j] << " ";
    cout << endl;
  }
```

```
  return 0;
}
```

5 Tableau dynamique dans une structure

Exemple de tableau dynamique dans une structure :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  typedef struct {
    double* x;
  } struty;
  int i, n = 5;
  struty z;
  z.x = (double*)malloc(n * sizeof(double));
  for(i = 0; i < n; i++)
    z.x[i] = i + 1.5;
  for(i = 0; i < n; i++)
    cout << z.x[i] << " ";
  cout << endl;
  return 0;
}
```

6 Tableau dynamique de structures

Exemple de déclaration d'un tableau dynamique de structures :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  typedef struct {
    int i;
    double x, y;
    char c;
  } struty;
  int n = 5;
  struty* w = (struty*)malloc(n * sizeof(struty));
  // On a alors n structures du type struty : w[0] ... w[k] ... w[n-1] avec k=0 ... n-1
  // Les membres s'obtiennent par : w[k].i, w[k].x, w[k].y, w[k].c
  return 0;
}
```

13. Fichiers sources, compilation séparée, visibilité

1 Introduction

Les fichiers contenant les instructions des programmes C sont nommés « fichiers sources » . Un programme[60] peut, ou non, être contenu dans un fichier source unique. Lorsqu'une même fonction est utilisée par des programmes différents, il est pratique de la mettre dans un fichier à part, ce qui évite d'avoir à la recopier dans le fichier de chaque programme l'utilisant. Un programme sera alors réparti dans plusieurs fichiers sources. Cette façon de procéder permet aussi :

— de rendre une variable globale entre certaines fonctions seulement et non entre toutes
— de faire qu'une fonction puisse être connue de certaines fonctions seulement et non de toutes.

2 Programme réparti dans plusieurs fichiers sources

Dans la suite `type` symbolisera un type quelconque (`int`, `double`, ...) et `x` une variable quelconque.
Soit par exemple le programme suivant inclus dans un fichier unique nommé *source_1.cpp* :

```
#include<iostream>
using namespace std;
//-------------------------
type f(type x) {            /* en-tete de la fonction f */
  ...
}
//-------------------------
int main() {
  f(...);                   /* appel de la fonction f */
}
```

Il comprend une fonction et le `main`. On peut placer la fonction dans un fichier à part, nommé par exemple *f.cpp*, qui contient uniquement :

```
type f(type x) {
  ...
}
```

Le programme principal est placé seul dans un autre fichier nommé *source_2.cpp* :

```
#include<iostream>
using namespace std;
type f(type x);
//-------------------------
int main() {
  f(...);
}
```

Il est identique à celui contenu dans *source_1.cpp* mais la déclaration[61] de la fonction est maintenant obligatoire. Quand on travaillera avec plusieurs fichiers sources, contrairement à ce qui a été fait jusqu'à présent avec un fichier source unique, on écrira explicitement tous les prototypes de fonctions.[62] Pour simplifier l'application de cette règle et éviter des redites on utilise la directive `#include`.
De façon générale, la directive `#include "mon_fich.h"` a pour effet d'insérer, avant la compilation, le contenu du fichier *mon_fich.h* par exemple, là où elle se trouve.
On place le ou les prototypes dans un fichier à part, qu'on nomme, par exemple, *proto.h*, qui, dans le cas de l'exemple précédent, contient uniquement :

```
type f(type x);             /* prototype de f */
```

et, dans le fichier contenant le programme principal, seule la troisième ligne est modifiée :

60. C'est à dire des directives, un `main` et des fonctions.
61. C'est à dire le prototype. En fait pour un prototype il suffit d'écrire `type f(type);`, c'est à dire sans le nom `x`. Mais si la fonction a beaucoup d'arguments il est souvent plus lisible d'inclure également les noms.
62. On rappelle que, dans le cas d'un fichier source unique, le prototype n'est pas obligatoire pour les fonctions dont l'en-tête précède l'utilisation dans le fichier.

```
#include<iostream>
using namespace std;
#include "proto.h"
//-------------------------
int main() {
  f(...);
}
```

et tout se passe exactement comme si on avait mis la ligne :

```
type f(type x);
```

à la place de la ligne :

```
#include "proto.h"
```

Le programme est maintenant réparti dans trois fichiers au lieu d'un, c'est à dire :

source_2.cpp, *f.cpp* et *proto.h*

au lieu de :

source_1.cpp

La compilation doit être faite par :

g++ -lm f.cpp source_2.cpp

l'ordre n'ayant pas d'importance. [63]

Désormais, tout programme, placé dans un fichier nommé par exemple *source_qqc.cpp*, contenant la directive :

```
#include "proto.h"
```

et compilé par :

g++ -lm f.cpp source_qqc.cpp

peut utiliser la fonction `f`.
f.cpp et *source_qqc.cpp* sont d'abord compilés séparément, puis reliés entre eux.

Remarque

La différence entre l'utilisation de < > et de " " dans les `#include` est qu'avec " " le compilateur cherche le fichier *.h* d'abord dans le même répertoire que le répertoire du fichier *.cpp* qui contient le `#include` (normalement le répertoire courant), tandis qu'avec < > il cherche directement dans les répertoires système.

3 Visibilité des variables globales entre fichiers sources

3.1 Rappel dans le cas d'un fichier source unique

3.1.1 Variables locales

Toutes les variables déclarées :

à l'intérieur du `main`
à l'intérieur ou en argument muet des fonctions

sont locales.

3.1.2 Variables globales

Les variables déclarées en dehors du `main` et des fonctions sont globales pour toutes les fonctions (y compris le `main`) suivant la déclaration dans le fichier source. Une exception arrive lorsqu'une variable locale porte le même nom qu'une variable globale : cette dernière est alors invisible dans la fonction contenant la variable locale.

3.2 Cas de plusieurs fichiers sources

3.2.1 Variables locales

Rien n'est changé.

63. L'option *-lm* est seulement nécessaire si l'un (ou plusieurs) des fichiers utilise `math.h`.

3.2.2 Variables globales

Soit par exemple les deux fichiers sources :
source_f.cpp

```
#include<iostream>
using namespace std;
#include "proto.h"
type p, q;
//-------------------------
int main() {
  ...
}
//-------------------------
type f1(...) {
  ...
}
//-------------------------
type f2(...) {
  ...
}
...
```

et :
source_g.cpp

```
#include "proto.h"
type a, b;
type g1(...) {
  ...
}
type g2(...) {
  ...
}
...
```

les prototypes des fonctions `f1`, `f2`, ..., `g1`, `g2`, ..., étant placés dans le fichier `proto.h`.
Pour ce qui va être dit dans la suite, les deux fichiers sources *source_f.cpp* et *source_g.cpp* doivent être vus comme complètement symétriques. En effet l'un contient le `main` mais celui-ci peut être considéré comme une fonction ordinaire.
Les variables `p`, `q`, ... sont globales pour toutes les fonctions du fichier source *source_f.cpp*, mais pas pour les fonctions du fichier source *source_g.cpp*.
Les variables `a`, `b`, ... sont globales pour toutes les fonctions du fichier source *source_g.cpp*, mais pas pour les fonctions du fichier source *source_f.cpp*.
Si l'on veut que `a` soit globale pour l'ensemble des deux fichiers, il faut la redéclarer dans *source_f.cpp* par :

```
extern type a
```

et symétriquement si on veut que `p` soit aussi globale pour les deux fichiers.

Si l'on veut empêcher que `b`, par exemple, puisse être rendue globale dans *source_f.cpp* par un :

```
extern type b
```

on la déclare :

```
static type b
```

dans *source_g.cpp*.

Remarque

le mot clé `static` employé dans ce contexte a une signification différente de celle qu'il a quand on veut signifier qu'une variable locale est statique. En effet `b` n'est pas locale puisqu'elle est globale et donc déjà statique.

Ainsi, même s'il existe une variable globale nommée `b` dans un troisième fichier source qu'on a rendue globale avec *source_f.cpp* en la déclarant dans ce dernier par :

```
extern type b
```

elle n'a rien à voir avec la variable globale `b` du fichier *source_g.cpp* qui n'est globale qu'à l'intérieur de ce dernier et il n'y a donc pas de conflit.
On voit donc qu'on peut déclarer des variables, globales au sein d'un fichier source, mais locales par rapport à d'autres fichiers source. Cette possibilité permet d'écrire des fichiers sources disposant en leur sein des avantages des variables globales, tout en pouvant être utilisés comme des boîtes noires, c'est à dire en ayant uniquement connaissance de ce qui entre et de ce qui sort.

Remarques

Les variables globales sont dites « privées » par rapport aux fichiers sources dont elles ne sont pas connues et « publiques » par rapport aux autres.

On rappelle que, de toutes façons, les variables globales ne doivent être utilisées qu'en dernier recours, car en établissant des liens entre des fonctions, elles rendent plus difficile de contrôler les conséquences des appels de ces fonctions.

4 Visibilité des fonctions entre fichiers sources

On se place toujours dans le cadre de l'exemple vu à la section précédente. Contrairement au cas des variables globales, les fonctions sont, à priori, toutes connues les unes des autres, on dit qu'elles sont « publiques » . Cela peut être un inconvénient pour assurer l'indépendance des fichiers sources. Supposons, par exemple, qu'on veuille créer de nouvelles fonctions dans *source_f.cpp* ou dans un nouveau fichier source. Il faut s'assurer en particulier que ces nouvelles fonctions ne portent pas le nom de fonctions déjà existantes dans *source_g.cpp*. Si, dans *source_g.cpp* seule la fonction `g1` est appelée depuis l'extérieur cela représente une contrainte inutile en ce qui concerne les fonctions autres que `g1`. Pour simplifier cela, l'attribut `static` donne la possibilité de rendre certaines fonctions connues uniquement à l'intérieur de leur fichier source[64]. Il suffit d'écrire l'en-tête de chaque fonction concernée :

```
static type g2(...);
```

`g2` est alors dite fonction « privée » du fichier source *source_g.cpp*.

5 Méthode générale pour constituer une bibliothèque avec *Linux*

Soit, par exemple, les trois fonctions suivantes, écrites respectivement dans les fichiers *combi.cpp*, *binomiale.cpp* et *eq_3d.cpp* : [65]

combi.cpp :

```
/* Calcul de C(n,k), pas d'avertissement en cas de dépassement */
#include "ma_bibli.h"
int combi(int n, int k) {
  int c, j;
  if(k > n-k)
    k = n-k;
  for(c = 1, j = 1; j <= k; j++)
    c = c*(n-j+1)/j;
  return c;
}
```

binomiale.cpp :

```
/* Fournit la valeur de la loi binomiale pour une proba p, un nombre d'épreuves n
   et un nombre de réalisations k */
#include<math.h>
#include "ma_bibli.h"
double binomiale(double p, int n, int k) {
  int i;
  double b, q = 1.-p;
  b = pow(q,n);
```

64. Comme dans le cas des variables globales.
65. On peut éviter la répétition des `#include` autre que `#include "ma_bibli.h"` dans chaque fichier en les mettant dans `ma_bibli.h`.

```
  for(i = 1; i <= k; i++)
    b = b*(n-i+1)/i*p/q;
  return b;
}
```

eq_ 3d.cpp :

```
/* Calcule les racines de l'équation du troisième degré ax^3+bx^2+cx+d=0 */
#include<iostream>
#include<math.h>
#include "ma_bibli.h"
using namespace std;
static double rac(double x) {
  double y;
  y = pow(fabs(x),1./3.);
  if(x < 0.)
    y = -y;
  return y;
}
int eq_3d(double a, double b, double c, double d, double* x) {
  double r, s, t, p, q, u, v, phi, rr, rrr, rs3, dps3;
  dps3 = 2*M_PI/3;
  if(a == 0.) {
    cout << "Equation de degre < 3" << endl;
    return -1;
  }
  r = b/a; s = c/a; t = d/a; p = s-r*r/3; q = 2*r*r*r/27 - r*s/3 + t; rs3 = r/3;
  u = q*q/4 + p*p*p/27;
  if(u > 0) {
    x[0] = rac(-q/2+sqrt(u)) + rac(-q/2-sqrt(u)) - rs3;
    return 1;
  }
  if(u == 0.) {
    v = 2*rac(-q/2);
    x[0] = v-rs3; x[1] = x[2] = -v/2-rs3;
    return 2;
  }
  rr = sqrt(-p*p*p/27); rrr = 2*sqrt(-p/3); phi = acos(-q/2/rr);
  x[0] = rrr*cos(phi/3)-rs3;
  x[1] = rrr*cos(phi/3+dps3)-rs3;
  x[2] = rrr*cos(phi/3-dps3)-rs3;
  return 3;
}
```

On veut mettre ces fonctions dans une bibliothèque, utilisable dans tout programme.
On place les trois fichiers *combi.cpp*, *binomiale.cpp* et *eq_ 3d.cpp* dans un répertoire quelconque, dont le nom complet est par exemple */home/lulu/mes_fonctions*. Dans ce même répertoire on écrit le fichier suivant, nommé obligatoirement *Makefile* :

```
CC=-c -Wall --pedantic
ma_bibli.ar : combi.o binomiale.o eq_3d.o
        ar -r ma_bibli.ar combi.o binomiale.o eq_3d.o
combi.o : combi.cpp
        g++ $(CC) combi.cpp
binomiale.o : binomiale.cpp
        g++ $(CC) binomiale.cpp
eq_3d.o : eq_3d.cpp
        g++ $(CC) eq_3d.cpp
```

Attention : au début des lignes commençant par `ar` et `g++` il ne faut pas mettre des blancs mais un caractère de

tabulation (appuyer une fois sur la touche *Tab*).
Toujours dans le même répertoire écrire le fichier nommé *ma_bibli.h* contenant les prototypes de chaque fonction :

```
#ifndef MA_BIBLI_H
#define MA_BIBLI_H
int combi(int, int);
double binomiale(double, int, int);
int eq_3d(double, double, double, double, double*);
#endif
```

Ensuite dans ce même répertoire taper la commande :

make

Toutes les fonctions sont alors compilées et, s'il n'y a pas d'erreur à la compilation, un fichier *ma_bibli.ar* est créé. La bibliothèque est alors prête à l'emploi. Pour utiliser ces fonctions dans un programme quelconque nommé *mon_prog.cpp* placé dans n'importe quel répertoire il faut y mettre la directive `#include<ma_bibli.h>` et compiler par :

g++ -lm -Wall -I/home/lulu/mes_fonctions mon_prog.cpp /home/lulu/mes_fonctions/ma_bibli.ar

(c'est ce qui est fait dans la commande *ccc*).
Si on modifie une fonction déjà existante de la bibliothèque il faut refaire *make*, cela ne recompilera que la fonction modifiée.
Si on ajoute une fonction il faut compléter les fichiers *Makefile* et *ma_bibli.h* puis faire *make*, seule la nouvelle fonction sera compilée.
Toute fonction de la bibliothèque peut appeler toute autre fonction de cette bibliothèque.