Quelques Idées d'utilisation du C++

Frédéric Hecht, Cours Informatique de Base 2015-2016, Master 1
Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie

22 octobre 2015

Table des matières

1 L'ordinateur

1.1 Comment ça marche

1.2 Les nombres dans l'ordinateur

(très fortement inspiré de wikipedia)

1.2.1 Les entiers

Dans un ordinateur, le Codage binaire est utilisé. Pour trouver la représentation binaire d'un nombre, on le décompose en somme de puissances de 2. Par exemple avec le nombre dont la représentation décimale est 59 :

$$\begin{aligned} 59 &= 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\ 59 &= 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ 59 &= 111011 \quad \text{en binaire} \end{aligned}$$

Avec n bits, ce système permet de représenter les nombres entre 0 et $2^n - 1$,donc un nombre k l'ecrit comme

$$k = \sum_{i=0}^{n-1} b_i 2^i, \qquad b_i \in \{0, 1\}$$

Représentation des entiers négatifs Pour compléter la représentation des entiers, il faut pouvoir écrire des entiers négatifs. On a introduit la représentation par complément à deux. Celle-ci consiste à réaliser un complément à un de la valeur, puis d'ajouter 1 au résultat. Par exemple pour obtenir -5 :

000101 codage de 5 en binaire sur 6 bits

111010 complément à un

111011 on ajoute 1 : représentation de -5 en complément à deux sur 6 bits

Avec n bits, ce système permet de représenter les nombres entre -2^{n-1} et $2^{n-1} - 1$, où tous les calculs se font dans l'anneau $\mathbb{Z}/2^n\mathbb{Z}$, et les nombres négatifs ont le bit de poids fort (b_{n-1}) égal à 1.

1.2.2 Les réels

Les nombres à virgule flottante sont les nombres des valeurs réels, ce sont des approximations de nombres réels. Les nombres à virgule flottante possèdent un signe s (dans $\{-1, 1\}$), une mantisse entière m (parfois appelée significande) et un exposant e. Un tel triplet représente un réel $s.m.be$ où b est la base de représentation (parfois 2, mais aussi 16 pour des raisons de rapidité de calculs, ou éventuellement toute autre valeur). En faisant varier e, on fait « flotter » la virgule décimale. Généralement, m est d'une taille fixée. Ceci s'oppose à la représentation

dite en virgule fixe, où l'exposant e est fixé. Les différences de représentation interne des nombres flottants d'un ordinateur à un autre obligeaient à reprendre finement les programmes de calcul scientifique pour les porter d'une machine à une autre jusqu'à ce qu'un format normalisé soit proposé par l'IEEE.

Les flottants IEEE peuvent être codés sur 32 bits (« simple précision ») ou 64 bits (« double précision »). La répartition des bits est la suivante :

	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^s \times (1, M) \times 2^{E-127}$
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^s \times (1, M) \times 2^{E-1023}$
Grande précision	80 bits	1 bit	15 bits	64 bits	$(-1)^s \times (1, M) \times 2^{E-16383}$

Précautions d'emploi

Exemple de la programmation de la série u_n des 1,

$$u_0 = 0, \qquad u_n = u_n + 1,$$

si les u_n sont des nombres flottants, il existe un grand nombre flottant K telle que 1. soit negligable par rapport K donc $K + 1$ est égal à K, comme la suite est croissante et majorée, elle est donc convergente.

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

int main (int argc,const char **)
{

  float x=0,x1=1;
  while( x < x1)
    {
      x=x1;
      x1=x+1;
    }
  cout << " lim de series des 1 en (float) =" << x << endl;
  return 0;
}
```

et le résulat est `x = 1.67772e+07` obtenu en moins d'une seconde car finalement, l'algorithme est en $O(1O^7)$ opération.

Exercice 1. Oui, mais quel est la loi du groupe $(\mathbb{R}, +)$ qui n'est pas vérifiée.

Exercice 2. Faire le même type de test avec les autre type de nombre : `int, short, char, long, long, long, double`

Les calculs en virgule flottante sont pratiques, mais présentent trois désagréments :

— leur précision limitée, qui se traduit par des arrondis qui peuvent s'accumuler de façon gênante. Pour cette raison, les travaux de comptabilité ne sont pas effectués en virgule flottante, même pour la fortune personnelle de Bill Gates, car tout doit y tomber juste au centime près.

— une quantisation qui peut être gênante autour du zéro : une représentation flottante possède une plus petite valeur négative, une plus petite valeur positive, et entre les deux le zéro... mais aucune autre valeur ! Que rendre comme résultat quand on divise la plus petite valeur positive par 2 ? 0 est une réponse incorrecte, garder la valeur existante une erreur aussi.
— des effets de « bruit discret » sur les trois derniers bits lorsque l'exposant est une puissance de 16 et non de 2

Il est par exemple tentant de réorganiser des expressions en virgule flottante comme on le ferait d'expressions mathématiques. Cela n'est cependant pas anodin, car les calculs en virgule flottante, contrairement aux calculs sur les réels, ne sont pas *associatifs*. Par exemple, dans un calcul en flottants IEEE double précision, $(10^{50} + 1) - 10^{50}$ ne donne pas 1, mais 0. La raison est que $10^{50} + 1$ est approximé par 10^{50}. Une configuration binaire est en général réservée à la représentation de la « valeur » NaN (« not a number »), qui sera par exemple le résultat de la tentative de division flottante d'un nombre par zéro. NaN combiné avec n'importe quel nombre (y compris NaN), donne NaN.

2 INITIATION AU SYSTÈME UNIX

2.1 UNE INTRODUCTION :

Pour utiliser un ordinateur, l'usager dialogue avec un logiciel qui gère l'ensemble de ses ressources (clavier, écran, souris, mémoire centrale, disques, imprimante, ...). Ce logiciel porte le nom de SYSTÈME D'EXPLOITATION. Le système d'exploitation des micro-ordinateurs PC peut être WINDOWS-XP, Linux ; ... . Celui des micro-ordinateurs APPLE s'appelle MacOS.

Ce système UNIX a été conçu à l'origine par Dennis RITCHIE et Ken THOMPSON des laboratoires Bell au début des années 1970. Depuis, il a donné naissance à une famille de systèmes (AIX, BERKELEY, LINUX, SYSTEM V, ...) qui "réagissent" comme UNIX. Chaque système UNIX permet le dialogue avec l'usager selon un langage de commandes et gère les ressources informatiques grâce à

— un système hiérarchisé de fichiers ;
— un gestionnaire de processus (séquence de commandes à exécuter) ;
— des fonctions spécialisées .

Les commandes permettent :

— toutes les manipulations de fichiers (ls, cp, mv, rm, ...) ;
— l'édition et le traitement de textes : (ed, vi, emacs, nedit, xedit, ...) ;
— la compilation des différents langages informatiques (C, C++, Fortran, Pascal, Ada, Java, ...) ;
— l'édition de liens des modules issus des compilations et leur archivage ; (ld, libtool, ar, ...)
— l'exécution des modules exécutables (NomFichier, ...) ;
— l'échange de données sur les réseaux de télécommunications tel INTERNET (firefox, scp, ssh, X11 ...).

L'interpréteur de commandes, encore dit le SHELL, peut exécuter des commandes regroupées

dans un fichier. L'ensemble de ces nouvelles commandes définit alors une nouvelle commande (de nom le nom du fichier) et ainsi de suite... Chaque utilisateur construit progressivement son environnement personnel pour répondre à l'ensemble de ses besoins.

2.2 LES GRANDS PRINCIPES D'UNIX :

2.2.1 LE SYSTÈME de GESTION des FICHIERS

Un fichier est formé d'informations (texte, instructions, commandes, images, sons, ...) regroupées sous un NOM et généralement conservées entre 2 exécutions. À ce fichier sont associés des DROITS D'ACCÈS qui permettent de l'utiliser en lecture et/ou en écriture et/ou en exécution. Pour éviter que les fichiers n'apparaissent sur un seul niveau, la notion de RÉPERTOIRE ou DOSSIER (directory en anglais) permet la hiérarchisation des fichiers en une ARBORESCENCE. Répertoires (nœuds communs à plusieurs branches de l'arbre) et fichiers (feuilles de l'arbre) constituent un arbre dont la racine (le répertoire initial) a pour nom /

Le caractère / en début d'un nom de fichier ou répertoire désigne le disque par défaut de l'ordinateur. Souvent, il s'agit de l'unique disque "dur" de la station de travail ou de l'ordinateur.

De même que pour les fichiers, les DROITS D'ACCÈS d'un répertoire permettent ou non la lecture de ses fichiers ou sous-répertoires, l'ajout ou non de nouveaux fichiers ou sous-répertoires aux utilisateurs habilités. Les unités périphériques (clavier, souris, modems, ...) envoient en mémoire soit centrale notée MC, soit secondaire (disque ou clef USB) notée MS, des informations (caractères tapés au clavier, mouvement de la souris, ...) regroupées sous la forme d'un fichier dit d'ENTRÉE.

En sens inverse, des informations contenues en mémoire et regroupées sous la forme d'un fichier dit de SORTIE peuvent être transmises aux unités périphériques (affichage sur l'écran ou l'imprimante, transfert de la MC sur MS, envoie sur une ligne téléphonique, ...). De façon générale, les fichiers d'ENTRÉES-SORTIES ne sont pas conservés après leur transfert.

2.2.2 Les PROCESSUS ou TACHES

Tous les travaux de l'usager sont réalisés par des processus. Un PROCESSUS est l'exécution d'un programme particulier appelé "SHELL" qui est en fait un interpréteur de commandes. Une commande frappée est interprétée par le processus qui réserve tout ce qui est nécessaire à son exécution (mémoire centrale, fichiers, ...) et l'exécute.

Au démarrage du système, un processus est créé. Un processus, dit alors père, peut demander la création d'un nouveau processus dit processus fils. Un processus fils est créé par copie du processus père. Le processus père a la possibilité d'attendre la fin de l'exécution du processus fils. Un processus peut aussi se remplacer lui-même par un autre code exécutable.

Un processus peut être exécuté en interactif ou en tâche de fond (& doit être ajouté à la fin de la commande).

Les commandes peuvent être tapées au clavier ou lues dans un fichier. Le nom d'un fichier contenant des commandes exécutables apparaît ainsi comme une nouvelle commande. Il est

ainsi possible de créer un environnement adapté aux besoins. Il est aussi possible d'enchaîner des processus, les résultats du précédent servant de données au suivant grâce à la notion de TUBE ou CANAL (pipe en anglais).

2.2.3 Les RESEAUX/ courrier

Les réseaux sous unix sont principalement de type **IP** (Internet Protocol). Le réseau est d'un point de vue information des découpé est 6 couches, la couche 1 correspondant a la partie support « hardware »(le type de fils paire torsadés, ondes radio, fibres optiques, IRD, ...), jusqu'à la couche logiciel finale correspondant au programme utilisé, par exemple : netscape, ssh, ...

De faite pour utiliser le réseau IP, il faut connaître le nom de la machine de l'on veut atteindre par exemple `www.yahoo.fr` ou `ftp.jussieu.fr` et quel type de service vous voulez utiliser

```
ftp 21/tcp
telnet 23/tcp
smtp 25/tcp mail
nameserver 42/tcp name          # IEN 116
whois 43/tcp nicname
bootp 67/udp                            # bootp server
finger 79/tcp
pop2 109/tcp postoffice
pop3 110/tcp pop
nntp 119/tcp readnews untp # USENET News Transfer Protocol
ntp 123/udp
snmp 161/udp
exec 512/tcp
biff 512/udp comsat
login 513/tcp
who 513/udp whod
shell 514/tcp cmd          # no passwords used
syslog 514/udp
printer 515/tcp spooler          # line printer spooler
talk 517/udp
ntalk 518/udp
route 520/udp router routed
timed 525/udp timeserver
```

Si vous voulez utiliser l'un de ces services il faut utiliser un client sur votre ordinateur (par exemple : ftp ftp.inria.fr) qui va envoyer une requête sur la machine ftp.inria.fr sur le port ftp, là le client ftpd qui tourne sur la ftp.inria.fr répondra, et vous pourrez vous connecter.

Quelques mots clefs, le DNS (domain name server) la partie logiciel qui permet de trouver l'adresse IP (4 octets noté a.b.c.b, où a,b,c et d sont des nombre entier compris entre 0 et 255, exemple : 134.157.2.1) d'une machine à partir du nom de la machine (exemple : emeu.ann.jussieu.fr).

Des ordinateurs serveurs DNS du réseau `jussieu.fr` ont pour numéro IP `134.157.2.1`, `134.157.0.129`, `134.157.8.84` .

Le routeur est un composant qui permet de passer d'un réseau à un autre. Le routeur du laboratoire d'analyse numérique est `134.157.2.254`, ce routeur est aussi un serveur de temps via le protocole `ntp` (net time protocole).

2.2.4 le courrier électronique

Le courrier électronique est quelque chose d'assez compliqué. Dès que vous avez un compte sur une machine UNIX vous disposez d'une boîte aux lettres sur cette machine si le service sendmail (smtp) est activé. Le problème est que généralement vous aurez des comptes sur chacune des machines de la salle informatique. Pratiquement, tous les courriers arriveront sur une machine unique que l'on appelera serveur, et donc généralement, vous allez vouloir lire votre courrier d'une autre machine. Pour cela il faudra utiliser un service IP pour lire courrier sur le serveur (POP3 ou IMAP) ou bien sinon il faudra vous connecter effectivement sur le serveur via `ssh`. Maintenant, il faut définir la machine qui va envoyer le courrier (généralement votre machine, cette machine doit être sur le même reseau (à jussieu cette machine peut être `shiva.ccr.jussieu.fr`).

Pour lire votre courrier électronique, vous pouvez utiliser `firefox, mail.app, ...` (les commandes mail, mailx, Mail sont fortement déconseillées car elles ne comprennent pas les attachements, le metamail (mime), les caractères accentués).

Si possible utiliser le protocol APOP (plutôt que POP) qui crypter votre mot de passe au moment de votre authentification sinon votre mot de passe passera en clair sur le réseau.

2.2.5 des commandes réseaux

Ici le mot « computer »sera un nom de machine du réseau, par exemple : `ibm1.cicrp.jussieu.fr` ou un numero IP `128.93.16.18`.

Version sécurisée, les informations qui passent sur le réseau, sont cryptées.

- `ssh nom_de_machine -l user` connection a un machine, version international (la clef de cryptage est de 1024 bits)
- `scp [-R] user@computer:name1 name2` copiez un fichier ou un directory de l'utilisateur `user` du `computer` dans `name2` sur votre ordinateur, avec l'option -R le directory est copié récursivement. Remarque le mot de passe est demandé à chaque utilisation.
- `scp [-R] name2 user@nom_de_machine:name1` réciproque.

Les commandes réseaux

- `ftp computer` ouvre un session de transfért de fichier sur la machine computer,
- `firefox` ouvre un broutteur de page `html`, de `news`, et lecteur de courrier électronique.

Les commandes pour connaître l'état du réseau.

- `ping computer` test le réseau entre vous et la machine `computer` , taper `ctrl C` pour arrêter.
- `nslookup computer` recherche le numéro IP d'un machine
- `netstat -nr` imprime toutes les routes connues par la machine

2.3 COMMENT UTILISER UNIX EN INTERACTIF :

Un utilisateur d'Unix est reconnu par son User-Id (par exemple p6dean12). De plus, il appartient à un ou plusieurs groupes repérés par un Group-ID (par exemple p6dean).

Ces 2 identificateurs ont dû être enregistrés par l'administrateur du système avant toute tentative de connexion.

La commande `id` ou `id UserId` retourne les 2 noms déclarés.

La commande `newgrp AutreGroupe` permet le changement de groupe d'un utilisateur.

2.3.1 LA CONNEXION

Après avoir allumé l'ordinateur, une fenêtre s'ouvre sur l'écran avec 2 cases : *Nom de l'Usager* et *Mot de Passe*.

À l'aide de la souris, le curseur doit être positionné dans la première case.

Ensuite, il faut taper votre nom d'usager et le valider par la frappe de la touche `<Entrée>`. Ce nom doit être connu du système, c'est à dire avoir été enregistré par l'administrateur du système.

Si la frappe est incorrecte ou bien si le nom est inconnu, il apparaîtra le texte « login incorrect »

et vous n'aurez plus qu'à recommencer...

Dans le cas contraire, votre mot de passe (8 caractères sont souhaités) doit ensuite être frappé et validé par la frappe de la touche `<Entrée>`. L'affichage de ces caractères est annihilé pour des raisons évidentes de sécurité. Une fenêtre s'ouvre sur votre écran. Derrière le curseur, il

est alors possible de taper une commande UNIX ... Pour terminer la session, il faut cliquer en

dehors de toute fenêtre, choisir l'option Fin de Session et confirmer la demande de sortie de mwm.

2.3.2 Le CHANGEMENT de MOT DE PASSE

(Obsolete car les password réseau gère par ldap sont a chanter sur le web.)

Il faut taper

`passwd` NomUsager.

Il apparaîtra ensuite

```
Old password:   (donnez l'ancien mot de passe)
New password:   (donnez le nouveau mot de passe)
Retype new password: (redonnez le meme nouveau mot de passe!).
```

ATTENTION : N'oubliez pas ce nouveau mot de passe sous peine d'avoir à relancer la procédure d'octroi d'un nouveau mot de passe ce qui demande du temps.

De plus, la législation réprime tout accès ou tentative d'accès de personnes non autorisées à un ordinateur. Il est **absolument nécessaire de protéger** votre **mot de passe** sous peine de risquer des poursuites dues à l'utilisation frauduleuse de votre nom par une tierce personne. Votre mot de passe équivaut à votre signature. Il est donc recommandé de choisir un mot difficile à deviner et surtout de ne jamais stocker ce nom sur papier. A vous de le mémoriser !

2.3.3 Les FENÊTRES sur l'ÉCRAN et les TOUCHES du CLAVIER

Dans tout ce qui suit, un terme entre < > désigne une touche du clavier.

Le caractère haut d'une touche du clavier s'obtient en gardant enfoncée la touche `<Shift>` et en frappant la touche.

Le caractère bas à droite d'une touche (par exemple : \ ...) s'obtient en gardant enfoncée la touche `<AltGr>` et en frappant la touche.

`<Backspace>` efface le caractère qui précède le curseur qui recule d'un caractère

`<Char Del>` efface le caractère du curseur sans bouger de place

Attention :

- Toute ligne est à terminer (valider) par la frappe de la touche `<Entrée>` ;

- Pour taper une commande sur plusieurs lignes, il faut terminer chaque ligne, exceptée la dernière, par un caractère \, SANS AUCUN AUTRE CARACTÈRE DERRIÈRE, et surtout pas un caractère "blanc". En effet, le caractère \ neutralise le caractère de fin de ligne <Entrée> placé derrière. S'il existe un caractère "blanc" invisible, \ le supprime et non pas celui de fin de ligne et cela entraîne une erreur car les arguments de la commande sur les lignes suivantes ne sont pas transmis.

Une FENÊTRE ouverte sur l'écran comprend plusieurs parties, une barre d'en tête, et une partie centrale.

Avec une sourie à 3 boutons :

- le bouton gauche, fait apparaître un menu pour gérer la fenêtre, notamment, la déplacer ou la détruire.

- le bouton droite réduit la fenêtre à une icône. (Il suffit pour régénérer la fenêtre à partir de l'icône, de cliquer 2 fois rapidement l'icône).

- le bouton milieu à l'extrême droite augmente la taille de la fenêtre jusqu'à lui faire occuper tout l'écran. + Une fenêtre avec 2 parties :

  — la partie basse ou ligne courante permet l'entrée des COMMANDES UNIX
  — la partie haute contient les RÉSULTATS des commandes déjà exécutées

La taille et la position de la fenêtre peuvent être modifiées comme suit :

  — - positionner avec la souris le curseur dans un des 4 coins du cadre de la fenêtre,
  — - appuyer sur le bouton gauche de la souris et déplacer la souris sans relâcher le bouton,
  — - relâcher le bouton à l'endroit choisi.

Pour faire apparaître une fenêtre cachée, il faut déplacer la fenêtre qui la masque.

`<Ctrl>D` sur la ligne courante d'une fenêtre détruit la fenêtre (ferme le shell).

Pour créer une nouvelle fenêtre (en fait, un processus qui exécute la commande xterm !), enfoncer la touche droite ou gauche de la souris en dehors de toute fenêtre et après déplacement cliquer l'option `Nouvelle Fenêtre`. Quelques instants plus tard, une nouvelle fenêtre s'ouvre. Une autre méthode consiste à taper l'instruction

```
xterm -sb -sl 10000 -cr red -title NomTitre \
-bg black -fg white -fn 9x15 -fb 9x15bold -geometry 80x50+0+0 &
```

Alors une fenêtre, avec ascenseur, pouvant contenir au plus 10000 lignes, avec un curseur rouge, un titre NomTitre, un fond noir et le texte écrit en blanc selon la fonte normale ou grasse formée de 9 caractères de large sur 15 de haut avec 50 lignes de 80 caractères s'ouvre en haut et à gauche sur l'écran. D'autres fontes sont très souvent disponibles (7x14, 8x13, 10x20, ...).

2.3.4 Les NOMS de FICHIERS et RÉPERTOIRES

Le nom du disque serveur de référence est noté `/` et sert de racine de l'arbre. Tout fichier (file) ou répertoire (directory) a un NOM dans l'arbre qui l'identifie. Un nom de fichier peut comprendre un SUFFIXE séparé du nom par un `.`

exemples : sp.f `/home/p6dean/tri.f` `/users/dupont/.profile`

Ce dernier est un fichier exécuté lors de l'initialisation de la session interactive.

Le suffixe suit des conventions : .f ou .F pour fortran, .c pour C, .pas pour pascal, .tex pour TEX, ... `/usr/local/mefisto` est un répertoire c'est à dire en fait la liste des noms de ses sous-répertoires et de ses fichiers. `/sbin` contient la plupart des commandes UNIX de base pour le démarrage de l'ordinateur ;

`/dev` contient les fichiers d'emploi des périphériques (lecteurs de bandes magnétiques, "drivers", ...) ;

`/tmp` contient les fichiers temporaires nécessités par le système ;

`/etc` contient des utilitaires d'administration du système ;

`/usr` contient des répertoires d'utilitaires pour les utilisateurs

`/usr/bin` contient la plupart des commandes UNIX ;

`/usr/man` contient les 8 chapitres de la documentation en ligne (cf man) ;

`/usr/lib` contient les librairies X, Motif, ... ;

`/home` contient les répertoires des Utilisateurs. À l'initialisation de la session interactive, une position dans l'arbre est fixée. Il s'agit du RÉPERTOIRE PERSONNEL (HOME directory) de l'usager. Si son nom est dupont alors son répertoire personnel a pour nom /home/p6dean/dupont

Ce répertoire est aussi, à cet instant, le RÉPERTOIRE COURANT, encore dit répertoire de travail (WORKING Directory).

De là, pour atteindre un fichier ou répertoire (navigation dans l'arbre), il faut taper

- soit, le NOM ABSOLU qui débute par la racine ;

par exemple : `/home/p6dean/dupont/.profile`

- soit, le NOM RELATIF au répertoire courant ;

par exemple : `.profile`

désigne le même fichier si le répertoire courant est le HOME directory. `.` désigne le nom du répertoire courant par exemple : /home/p6dean/dupont

`..` désigne le nom du répertoire père du répertoire courant par exemple :

le répertoire père de `/home/p6dean/dupont` est `/home/p6dean`

2.3.5 LES PRINCIPALES COMMANDES D'UNIX

Une COMMANDE est constituée d'un NOM de COMMANDE suivi de 0 ou plusieurs mots séparés par au moins un caractère blanc ou tabulation.

`NomCommande {-option} {paramètre} {redirection}`

Les caractères { et } ne sont là que pour indiquer que le texte inclus peut être présent, répété ou absent.

(Dans les lignes qui suivent `fic` désigne un nom quelconque de fichier et `rep` un nom quelconque de répertoire) `ls` rep liste les fichiers et répertoires du répertoire rep

`ls -l` rep liste les fichiers et répertoires du répertoire rep avec ses droits d'accès et d'autres informations (taille, date de dernière utilisation, ...)

`ls -a` rep liste les fichiers et notamment ceux débutant par le caractère . (à ne pas confondre avec l'abréviation pour le répertoire courant) et les répertoires du répertoire rep avec d'autres informations.

`ls -R` rep liste les fichiers et sous-répertoires du répertoire rep mais aussi des sous-sous-...-répertoires du répertoire rep

`ls -rt` rep liste les fichiers et sous-répertoires du répertoire rep dans l'ordre inverse de la date de dernière modification. Le caractère > redirige les sorties.

`ls -1 >fic`

envoie les noms des fichiers et répertoires du répertoire courant, 1 par ligne, dans le fichier de nom fic et non pas sur l'écran ! Les sorties sont dites redirigées dans le fichier fic. Si le fichier fic existait son ancien contenu est perdu.

`ls -1 >>`fic effectue la même opération mais en ajoutant les noms à la suite du contenu actuel du fichier fic s'il existe. `cat` fic1 fic2 fic3 >fic concatène les fichiers fic1, fic2, fic3 dans le fichier fic.

En l'absence de >fic le fichier concaténé apparaît sur l'écran.

Attention de faite, il y a 2 types canaux de sortie sous unix le « standard output »et le « error output »qui ont respectivement comme numéro 1 et 2, si vous voulez stocker dans un fichier les erreurs des compilations il faut rediriger le canal 2 et non le canal 1 comme dans

`make essai 2>list.err`

Mais si vous voulez rediriger les deux canaux dans le canal 1 que vous pouvez ajouter `2>&1` , ce qui donne la commande suivante

```
make essai 2>&1 1>list.err
```

mais vous ne voyez plus rien, alors utilise la command tee qui duplique le standard input dans un fichier et dans le standard output, comme suit :

```
make essai 2>&1 | tee list.err
```

Pour plus de detail

```
man sh
```

`cat >fic` permet l'entrée au clavier du texte d'un fichier, à terminer par la frappe des touches <Ctrl> <D>.

Mais attention, il ne faut pas se tromper car à part la touche <Backspace>, rien n'est prévu pour faire des modifications (Pour cela, employer un éditeur de textes ed, vi, ...). De même, le

caractère < redirige les entrées. Par exemple

```
mail AdresseElectronique < lettre
```

envoie à l'adresse électronique le contenu du fichier lettre (de dernière ligne contenant en première colonne le caractère . et rien d'autre) du répertoire courant (obsolète) utiliser `pine` ou `netscape` .

Si le choix de l'une des options d'une commande vous pose des problèmes, il est possible de lire la documentation en ligne en tapant

`man NomCommande {numéro 1 à 8 du chapitre}`

Par exemple : man ls 1

Le texte de la documentation apparaît et il suffit de parcourir le texte en s'aidant des touches marquées d'une flèche (4 sens) pour obtenir les renseignements recherchés. Il existe 8 chapitres

dans cette documentation en ligne :

1 : Les commandes accessibles à l'usager

2 : L'interface entre UNIX et le langage C

3 : La bibliothèque C et macros et fonctions mathématiques

4 : Les caractéristiques des fichiers système associés aux périphériques

5 : Les formats des fichiers système

6 : Les jeux

7 : Les bibliothèques de macros pour la manipulation des documents

8 : Les commandes de maintenance du système

— `date` affiche l'heure et la date du jour.
— `cal 10 1999` affiche le calendrier du mois d'octobre 1999.
— `cd` rep rep devient le répertoire courant.
— `cd` le répertoire courant devient le répertoire personnel (cd $HOME).
— `mkdir` rep (Make Directory) crée dans le répertoire courant le nouveau répertoire rep
— `pwd` (Print Working Directory) affiche le nom du répertoire courant

— `rmdir` rep (Remove Directory) détruit le répertoire rep sous la condition qu'il soit vide de tout fichier et sous-répertoire
— `cp` fic1 fic2 ... ficn rep (Copy) fait une copie des fichiers fic1 fic2 ... ficn dans le répertoire rep sans détruire les fichiers fic1 fic2 ... ficn initiaux
— `mv` rep1 rep2 (Move) copie tous les fichiers et répertoires du répertoire rep1 dans le répertoire rep2, puis détruit le répertoire rep1
— `mv` fic1 fic2 ... ficn rep (Move) déplace les fichiers fic1 fic2 ... ficn dans le répertoire rep et détruit les fichiers fic1 fic2 ... ficn initiaux (GARE aux ERREURS si rep est oublié ! ficn contient fic n-1 et fic1 fic2 ... fic n-1 sont détruits !)
— `rm` fic (Remove) détruit le fichier fic
— `rm -R` rep (Remove) détruit toute la branche issue du répertoire rep et le répertoire rep lui-même. A utiliser avec discernement !
— `lpr -Plaser1` fic imprime le fichier fic sur l'imprimante de nom laser1.
— `more` fic affiche le contenu du fichier fic.
Pour passer à la page suivante taper un espace. Attention, pas de remontée possible.
Pour terminer, taper :q

Pour spécifier plusieurs noms de fichiers ou répertoires, il est possible d'employer la "STAR-CONVENTION" c'est à dire les caractères JOKER encore dits métacaractères (neutralisables par \) :
— `*` remplace n'importe quel nom jusqu'au prochain blanc ou . ou / possible
exemple :
— *.f désigne tous les fichiers du répertoire courant suffixés par f
— *.* désigne tous les fichiers ayant un suffixe
— `?` remplace n'importe quel caractère
— `!` indique NON de ce qui suit
exemple :
sp ?.f désigne tous les fichiers dont le nom commence par les 2 caractères sp, suivi d'un caractère et de suffixe .f .
— `[a-z]` remplace n'importe quel caractère a ou b ou ... ou z
— `[!a-w]` remplace n'importe quel caractère différent de a ou b ou ... ou w
— `[0-9]` remplace n'importe quel caractère 0 ou 1 ou ... ou 9
Avant utilisation, il est prudent d'afficher le résultat de l'analyse des ces caractères JOKER à l'aide de la commande echo :

`echo` NomAvecCaractèresJOKER affiche tous les noms ainsi désignés

Une commande peut souvent être stoppée par`<Ctrl> C`

2.3.6 Droits d'accès

Les `Droits d'accès` d'un fichier ou répertoire peuvent être listés par la commande `ls -l` et redéfinis par leur propriétaire à l'aide de la commande

`chmod -R rwxrwxrwx` rep ou fic

`chmod mode` rep ou fic

où r vaut 0 ou 4, w vaut 0 ou 2, x vaut 0 ou 1, sont à sommer pour donner l'autorisation en lecture (r=4), en écriture(w=2), en exécution (x=1), pour les 3 cas :

- le premier groupe rwx est pour le propriétaire ;

- le second groupe rwx est pour le groupe ;

- le troisième groupe rwx est pour tous les autres utilisateurs ;

- -R signifie récursivement c'est à dire dans les sous-répertoires aussi ;

- mode vaut

. en premier caractère `u` (utilisateur) ou `g` (groupe) ou `o` (autres) ou `a` (à la fois ugo)

. en second caractère + ou − ou = pour ajouter, supprimer ou définir une valeur ;

. en troisième caractère `r` pour lecture, `w` pour écriture et `x` pour exécution.

Exemples :

```
chmod 750 fic1
```

donne les droits de lecture-écriture-exécution au propriétaire, lecture-exécution (pas en écriture !) aux membres du même groupe et aucun accès aux autres utilisateurs du fichier fic1.

```
chmod o-w MonFichier
```

retire la possibilité d'écrire aux autres (excepté le propriétaire et les membres du même groupe) et donc de modifier le fichier MonFichier.

```
chmod a+rx MaProc
```

permet à tout le monde la lecture et l'exécution du fichier MaProc.

2.3.7 recherche des occurrences

La recherche des occurrences d'un MOT dans le contenu des fichiers d'une liste de fichiers (générés à partir de caractères JOKER) s'obtient par la frappe de

`grep` MOT NomsDesFichiers affiche les lignes contenant au moins une fois MOT dans le contenu des fichiers d'une liste de fichiers

`grep -n` MOT NomsDesFichiers affiche les lignes et leur numéro contenant au moins une fois MOT dans le contenu des fichiers NomsDesFichiers

`grep -i` MOT NomsDesFichiers affiche les lignes contenant au moins une fois MOT sans se préoccuper de l'aspect minuscule ou majuscule dans le contenu des fichiers NomsDesFichiers

NomsDesFichiers peut être défini avec des caractères JOKER. `find` rep -name MOT -print

affiche les noms des fichiers du répertoire rep dont chaque nom contient MOT . Attention, si vous voulez utilisez la "STAR-CONVENTION", il faut mettre entre quote "MOT" sinon le shell va interprète le mot comme plusieurs mots. Exemple pour trouver vous fichiers .f. `cd ; find . -name "*.f" -print` Ici aussi, MOT peut être décrit avec des caractères JOKER. `wc` fic donne le nombre de lignes, mots et caractères du fichier. `grep MOT *.f | wc`

équivaut à

```
grep MOT *.f >fic; wc fic; rm fic
```

Le caractère séparateur ; permet la frappe de plusieurs commandes sur une même ligne.

Le TUBE ou CANAL | permet d'employer les sorties de la première commande comme entrées de la seconde sans avoir à utiliser un fichier auxiliaire.

`diff` NomFichier1 NomFichier2 affiche toutes les différences entre les contenus des fichiers NomFichier1 et NomFichier2

2.3.8 archivage de fichiers

Le programme archiveur tar

Pour archiver des fichiers, on utilise le programme tar. Dont les principales options sont :

- **c** (Create) pour créer une archive
- **x** (eXtract) pour extraire les fichiers d'une archive
- **t** (lisT) pour afficher la liste des fichiers d'une archive
- **v** (Verbose) pour le mode verbeux
- **f** (Force) pour forcer le remplacement de fichiers
- **z** (gZip) traite les fichiers avec gzip (compression après archivage, décompression avant extraction et décompression temporaire pour afficher la liste des fichiers). Créer une archive
- **j** (bzip2) traite les fichiers avec bzip2 (compression après archivage, décompression avant extraction et décompression temporaire pour afficher la liste des fichiers). Créer une archive

Syntaxe : `tar cvf fichier.tar motif`

Exemple : `tar cvf tpc.tar *.c`

Dans l'exemple précédent, on crée une archive qui porte le nom tpc.tar qui contient tous les fichiers d'extention .c du répertoire courant.

Le motif est une expression régulière du Shell qui peut donc contenir des métacaractères.

Les fichiers archives doivent porter l'extention .tar.

Extraire les fichiers d'une archive

(Après avoir créé une archive) On remplace l'option c (create) par x (extract) pour extraire tous les fichiers d'une archive.

Syntaxe : `tar xvf fichier.tar`

Exemple : `tar xvf tpc.tar`

On peut n'extraire de l'archive que les fichiers satisfaisant un motif (encore une expression régulière).

Syntaxe : `tar xvf fichier.tar motif`

Exemple : `tar xvf tpc.tar poly*`

Dans l'exemple précédent, on extrait de l'archive tpc.tar seulement les fichiers dont le nom commence par poly.

Archive et compression automatique

Pour compresser automatiquement le fichier archive pendant sa création, on utilise l'option z (gzip).

Syntaxe : `tar zcvf fichier.tar.gz motif`

Exemple : `tar zcvf tpc.tar.gz *.c`

Et de façon similaire, pour décompresser une archive et en extraire les fichiers :

Syntaxe : `tar zxvf fichier.tar.gz`

Exemple :`tar zxvf tpc.tar.gz`

L'ordre des options n'a pas d'importance, en revanche, celui de fichier et motif en a.

Les fichiers compressés avec gzip ont .gz pour extention.

Contenu d'une archive

Pour visualiser la liste des noms des fichiers contenus dans une archive, on utilise l'option t.

Syntaxe : `tar tf fichier.tar`

Exemple : `tar tf tpc.tar`

Si le fichier est compressé avec gzip, on rajoute l'option z.

Syntaxe : `tar zt fichier.tar.gz`

Exemple : tar zt tpc.tar.gz !

2.3.9 Compression de données

`Les commandes GZIP et GUNZIP :`

Ces commandes compressent et décompressent le contenu du fichier donné. Le fichier compressé a pour nom son ancien nom suffixé par `.gz`

Par exemple :

`gzip rep.tar`

fournit le fichier compressé rep.tar.gz

Inversement,

`gunzip rep.tar.gz`

redonne le fichier rep.tar sous sa forme non compressée.

Au total, pour transporter un répertoire (et tous ses sous-répertoires), la commande tar est d'abord utilisée et immédiatement suivie de gzip.

Par exemple :

tar -cvf rep.tar rep

gzip rep.tar

Après transport et positionnement dans le bon répertoire

gunzip rep.tar.gz

tar -xvf rep.tar

Le taux de compression est parfois étonnant sur les fichiers textes, mais, moins important sur les fichiers binaires.Il est souvent possible de coupler tar et gunzip ou l'opération inverse en ajoutant simplement l'option z à celles de la commande tar.

Par exemple sous LINUX :

`tar -xzvf rep.tar.gz`

décompresse et copie dans les répertoires les différents fichiers.

2.3.10 Les LIENS SYMBOLIQUES :

La commande

`ln -s AncienNom NomLié`

permet de déplacer facilement un fichier ou un répertoire utilisé seulement selon son NomLié.

Exemple :

`ln -s /home/p6dean/p6dean00/MonLogiciel /usr/local/MonLogiciel`

permet l'écriture de procédures avec la variable d'environnement

`MonLogiciel=/usr/local/MonLogiciel`

mais aussi de placer ce logiciel n'importe où dans l'arbre des répertoires.

`ls -l NomLié`

redonne l'AncienNom véritable.

`rm NomLié`

ne détruit pas le fichier mais seulement le lien, c'est-à-dire que le NomLié n'existe plus.

2.4 UTILISATION DE L'ÉDITEUR vi :

Cet éditeur est toujours présent sous Unix et très utile lorsque X n'est pas actif. De plus, il sert à manipuler les commandes de l'historique de la connexion.

(Au CICRP, l'éditeur nedit (ou nedit) (taper `nedit &`) sera préféré car il emploie plus intensivement la souris et le clavier pour rectifier le texte, les menus pour éditer, sauvegarder, ouvrir les fichiers, obtenir une aide en ligne, ...) `vi NomFichier` charge le fichier et l'affiche sur l'écran

2 modes : le mode commande (pour déplacer le curseur ou modifier le texte) et le mode texte (pour sa saisie)

2.4.1 Sorties et sauvegardes

`:wq` ou `:x` sauvegarde du fichier sur lui-même et sortie de vi

`:q` sortie de vi

`:q!` sortie forcée de vi sans sauvegarde

`:w` sauvegarde du fichier sur lui-même

`:w AutreFichier` sauvegarde dans AutreFichier

2.4.2 Déplacements du curseur dans le texte (Pas de : devant)

`Ctrl-F` (Forward) déplacement d'un écran vers le bas

`Ctrl-D` (Down) déplacement d'un demi-écran vers le bas

`Ctrl-B` (Backward) déplacement d'un écran vers le haut

`Ctrl-U` (Up) déplacement d'un demi-écran vers le haut Ligne Courante c'est la ligne où se trouve le curseur `Espace` déplace le curseur d'un caractère vers la droite

`Entrée` passage à la ligne suivante

`+` passage à la ligne suivante

`−` passage au début de la ligne précédente

`/chaîne/` déplace le curseur au premier caractère de la première occurrence de chaîne rencontrée (Attention : minuscule et majuscule sont différenciées)

`?chaîne?` de même mais en remontant le texte

`G` déplace le curseur à la fin du texte

`w` déplace le curseur au début du prochain mot

`3w` déplace le curseur au début du 3-ème prochain mot

`b` déplace le curseur au début du mot précédent

`5b` déplace le curseur au début du 5-ème mot précédent

`3H` déplace le curseur au début de la 3-ème ligne de l'écran (et non du texte)

`4L` déplace le curseur au début de la 4-ème ligne avant la fin de l'écran ou du texte situé sur l'écran

`M` déplace le curseur au début de la ligne située au milieu de l'écran

`z.` centre l'écran sur la ligne courante

`~ sur une lettre` transforme la lettre majuscule en minuscule ou le contraire Selon le terminal utilisé (xterm, vt100,...), les touches "Flèches" peuvent ou non être employées pour déplacer le curseur d'un caractère dans les 4 directions ou d'un écran vers le haut ou le bas.

2.4.3 Suppressions de caractères

`x` supprime le caractère où se trouve le curseur

`4x` supprime les 4 caractères à partir du curseur

`dd` supprime la ligne support du curseur

`3dd` supprime la ligne courante et les 2 lignes suivantes

`D` supprime le reste de la ligne courante à partir du curseur

`dw` supprime le mot courant

2.4.4 Saisie de caractères => Passage en MODE TEXTE

`<Échappement>` provoque la sortie du mode texte et l'entrée en mode commande

`i` permet ensuite l'insertion continue de caractères juste avant le curseur et cela jusqu'à la frappe du caractère `<Échappement>`

`I` idem mais au début de la ligne courante

`a` permet ensuite l'insertion continue de caractères juste après le curseur et cela jusqu'à la frappe du caractère `<Échappement>`

`A` idem mais à la fin de la ligne courante

`C` remplace le reste de la ligne à partir du curseur

2.4.5 Déplacement de textes

La destruction précède le déplacement puis la restitution

Exemple :

/continue/

5dd

10-

p

Les 5 lignes de la prochaine occurrence de continue sont détruites

le curseur est remonté de 10 lignes

Le texte des 5 lignes est restitué à partir de la position du curseur

En fait, toute destruction est mémorisée, le curseur doit être déplacé et le texte mémorisé est restitué par la frappe de `p`

`9p` restitue les 9 dernières destructions (9 est le maximum permis) De même, `yy` mémorise la ligne (`5yy` mémorise la ligne courante et les 4 lignes suivantes) et après déplacement `p` les restituent. Ainsi, il est possible de copier une partie du texte. `1.4.6. Exécuter des requêtes (précédées du caractère :)`

`:début,fin p` affiche les lignes de numéro début à fin

début est un numéro de ligne ou . la ligne courante ou .+4 la 4-ème ligne après la ligne courante ou .-3 la 3-ème ligne avant la ligne courante

fin est un numéro de ligne ou . la ligne courante ou comme ci-dessus ou $ la dernière ligne du texte

sans précision de ligne, la ligne courante est utilisée

`:début,fin s/AncienneChaine/NouvelleChaine/`

cette requête substitue pour les lignes début à fin l' AncienneChaine par la NouvelleChaine

`u` restitue l'état antérieur des lignes avant la dernière substitution

Dans la définition d'une chaîne :

`.` désigne n'importe quel caractère

`*` désigne la répétition (voire 0 fois) du caractère précédent

`/.*;/` désigne n'importe quelle chaîne terminée par ;

(Dans le shell * suffit pour désigner ici .*)

`^` désigne le caractère fictif qui précède le premier de la ligne

`$` désigne le caractère fictif qui suit le dernier de la ligne

`[a-z]` désigne n'importe quelle lettre minuscule (une seule)

`[^ 0-9]` désigne tout caractère différent d'un chiffre 0 à 9

`^` désigne le complémentaire de l'ensemble des caractères qui suit

sauf s'il n'est pas le premier caractère et il désigne alors le caractère lui-même

`\` neutralise le caractère spécial (.\ $ * ^ [ ]) qui le suit

`:s/1\ .2\ ..*/3/` substitue 1.2. et le reste de la ligne par 3

`:s/a+3/(&)/` encadre de parenthèses les occurrences de a+3

`&` désigne la chaîne de caractères qui précède

`:/\ (chaîne1\)caractère ou non\(chaîne2\) .../`

définit 2 ou plusieurs sous-chaînes limitées par \(et \) et utilisables ensuite avec `\ 1` ou `\ 2`

Exemple :

abcdefg hijklmno pqrstuv wxyz

:s/\ (.* \ ) \ (.* \ ) \ (.* \ ) \ (.*\ )/\ 3 \ 1\ .\ 4 ;\ 2/

donne

pqrstuv abcdefg .wxyz ;hijklmno

`:début,fin g/chaîne/requête` fait subir à toute ligne contenant chaîne la requête qui suit

`:.,.5g /^ a/s/a/A/` substitue sur la ligne courante et ses 5 suivantes le premier caractère a de la ligne en A

Si plusieurs requêtes s'avèrent nécessaires, avant de frapper la touche entrée, il faut frapper le caractère \ pour annihiler la fin de ligne `v` en lieu et place de `g` applique les requêtes sur les lignes ne contenant pas chaîne

`:r fichier` ajoute le texte du fichier à partir de la position du curseur

`:!commande` exécute la commande Unix puis revient dans vi

Exemple : `:!cd /MonCatalogue`

2.4.6 Les options de vi

`:set all` liste toutes les options possibles

Une option peut être activée par

`:set NomOption` ou

`:set NomOption=valeur`

et supprimée par

`:set no NomOption`

Les plus intéressantes sont :

`autoindent` ou en abrégé `ai` autoindentation

`autowrite` ou `aw` sauvegarde automatique avant :n et :!

`ignorecase` ou `ic` pas de distinction entre minuscule et majuscule

`list` affiche les caractères de tabulation selon ^ I

`number ou nu` affiche le numéro de chaque ligne

2.5 LES VARIABLES et les FICHIERS de DÉMARRAGE

Attention ceci n'est valable que si vous utiliser un shell de type `sh,ksh, bash,` ou `zsh` et non `csh` ou `tsch`.

Les variables locales n'ont pas à être déclarées. Il suffit de les initialiser. Par exemple

`VarLocale1=valeur1 VarLocale2=valeur2 ...`

La `valeur` de VarLocale1 est `$`VarLocale1 ou `${`VarLocale1`}` ce qui en permet la concaténation avec tout autre texte. Pour rendre ces variables locales permanentes, il faut les "exporter" par la commande

`export` VarLocale1 VarLocale2

Dès lors, ces variables deviennent des variables d'environnement utilisables dans n'importe quelle procédure de commandes.

Certaines de ces variables sont définies par le système ou par l'exécution de fichiers de démarrage.

La frappe de la commande `env` liste toutes les variables d'environnement actuelles.

Par exemple :

- `HOME` contient le nom du répertoire personnel ;

- PATH contient la liste des répertoires où les commandes doivent être recherchées.

- CDPATH contient la liste des répertoires atteignables par la commande cd. Pour initialiser ou

modifier ces variables, ou en créer d'autres, il peut être judicieux d'ajouter des commandes aux fichiers de démarrage. Leur liste partielle s'obtient par la commande

ls -a il apparaît

```
.    ..    .profile  .bashrc  .kde
```

Le listage à l'aide de la commande cat des fichiers .profile et .kshrc donne Pour modifier son

environnement initial, il suffit de modifier le fichier .profile du répertoire personnel. Pour modifier son environnement lors de l'ouverture d'un processus ksh, il suffit de modifier le fichier .kshrc du répertoire personnel. Attention à la différence entre les 2 appels :

.profile est exécuté une seule fois avant appel de ksh ;

.kshrc est exécuté lors du démarrage à chaque appel de /bin/ksh .

1.6. Le mécanisme d'historique des commandes en Korn Shell Il faut d'abord

ajouter au fichier .profile la commande set -o vi

Les commandes frappées (au plus 128 par défaut de la variable HISTSIZE) sont alors stockées dans le fichier .sh_history. Ensuite, pour

- faire apparaître en ligne de commande la commande précédente, taper les touches <Échappement> et <k> ;

- faire apparaître en ligne de commande la commande suivante, taper les touches <Échappement> et <j> ;

- avancer d'un caractère sur la droite, taper les touches <Échappement> et <Espace> ou <l> ;

- avancer d'un caractère sur la gauche, taper les touches <Échappement> et <Backspace> ou <h> ;

- avancer à la fin de la ligne , taper les touches <Échappement> et <Shift> et <$> ;

- détruire un caractère, taper les touches <Échappement> et <x> ;

- insérer un caractère avant le curseur, taper les touches <Échappement> et <i> ;

- insérer un caractère après le curseur, taper les touches <Échappement> et <a> ;

- sortir du mode historique, taper les touches <Ctrl> et <C> .

Bref, l'emploi de l'historique suit les règles d'emploi de l'éditeur vi en ajoutant au préalable la frappe de la touche <Échappement>

2.6 Connexion au machine de l'ufr

3 Présentation du langages C

3.1 introduction

3.1.1 Historique

Dans les dernières années, aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de C et de son jeune frère C++. L'étonnant dans ce fait est que le langage C n'est pas un nouveau-né dans le monde informatique, mais qu'il trouve ses sources en 1972 dans les 'Bell Laboratories' : Pour développer une version portable du système d'exploitation UNIX, Dennis M. Ritchie a conçu ce langage de programmation structuré, mais « très près »de la machine.

K&R−C

En 1978, le duo Brian W. Kernighan / Dennis M. Ritchie a publié la définition classique du langage C (connue sous le nom de standard K&R−C) dans un livre intitulé 'The C Programming Language'.

ANSI-C

Le succès des années qui suivaient et le développement de compilateurs C par d'autres maisons ont rendu nécessaire la définition d'un standard actualisé et plus précis. En 1983, le 'American National Standards Institute' (ANSI) chargeait une commission de mettre au point 'une définition explicite et indépendante de la machine pour le langage C', qui devrait quand même conserver l'esprit du langage. Le résultat était le standard ANSI-C. La seconde édition du livre 'The C Programming Language', parue en 1988, respecte tout à fait le standard ANSI-C et elle est devenue par la suite, la 'bible' des programmeurs en C.

C++

En 1983 un groupe de développeurs de AT&T sous la direction de Bjarne Stroustrup a créé le langage C++. Le but était de développer un langage qui garderait les avantages de ANSI-C (portabilité, efficience) et qui permettrait en plus la programmation orientée objet. Depuis 1990 il existe une ébauche pour un standard ANSI-C++. Entre-tempsAT&T a développé deux compilateurs C++ qui respectent les nouvelles déterminations de ANSI et qui sont considérés comme des quasi-standards (AT&T-C++ Version 2.1 [1990] et AT&T-C++ Version 3.0 [1992]), la version c++11 ISO/IEC 14882 :[2011].

3.1.2 Avantages

Le grand succès du langage C s'explique par les avantages suivants ; C est un langage :

(1) **universel :** C n'est pas orienté vers un domaine d'applications spéciales, comme par exemple FORTRAN (applications scientifiques et techniques) ou COBOL (applications commerciales ou traitant de grandes quantités de données).

(2) **compact :** C est basé sur un noyau de fonctions et d'opérateurs limité, qui permet la formulation d'expressions simples, mais efficaces.

(3) **moderne :** C est un langage structuré, déclaratif et récursif ; il offre des structures de contrôle et de déclaration comparables à celles des autres grands langages de ce temps (FORTRAN, ALGOL68, PASCAL).

(4) **près de la machine :** comme C a été développé en premier lieu pour programmer le système d'exploitation UNIX, il offre des opérateurs qui sont très proches de ceux du langage machine et des fonctions qui permettent un accès simple et direct aux fonctions internes de l'ordinateur (p.ex : la gestion de la mémoire).

(5) **rapide :** comme C permet d'utiliser des expressions et des opérateurs qui sont très proches du langage machine, il est possible de développer des programmes efficients et rapides.

(6) **indépendant de la machine :** bien que C soit un langage près de la machine, il peut être utilisé sur n'importe quel système en possession d'un compilateur C. Au début C était surtout le langage des systèmes travaillant sous UNIX, aujourd'hui C est devenu le langage de programmation standard dans le domaine des micro-ordinateurs.

(7) **portable :** en respectant le standard ANSI-C, il est possible d'utiliser le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant.

(8) **extensible :** C ne se compose pas seulement des fonctions standard ; le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

3.1.3 Désavantages

Évidemment, rien n'est parfait. Jetons un petit coup d'oeil sur le revers de la médaille :

(1) **efficience et compréhensibilité :** En C, nous avons la possibilité d'utiliser des expressions compactes et efficientes. D'autre part, nos programmes doivent rester compréhensibles pour nous-mêmes et pour d'autres. Comme nous allons le constater sur les exemples suivants, ces deux exigences peuvent se contredire réciproquement.

Exemple 1

Les deux lignes suivantes impriment les N premiers éléments d'un tableau A[], en insérant un espace entre les éléments et en commençant une nouvelle ligne après chaque dixième chiffre :

```
for (i=0; i<n; i++)
     printf("%6d%c", a[i], (i%10==9)?'\n':' ');
```

Cette notation est très pratique, mais plutôt intimidante pour un débutant. L'autre variante, plus près de la notation en Pascal, est plus lisible, mais elle ne profite pas des avantages du langage C :

```
for (I=0; I<N; I=I+1)
     {
      printf("%6d", A[I]);
      if ((I%10) == 9)
          printf("\n");
```

```
    else
        printf(" ");
  }
```

Exemple 2

La fonction copietab() copie les éléments d'une chaîne de caractères T[] dans une autre chaîne de caractères S[]. Voici d'abord la version 'simili-Pascal' :

```
void copietab(char S[], char T[])
{
  int I;
  I=0;
  while (T[I] != '\0')
   {
     S[I] = T[I];
     I = I+1;
   }
  S[I] = '\0';
}
```

Cette définition de la fonction est valable en C, mais en pratique elle ne serait jamais programmée ainsi. En utilisant les possibilités de C, un programmeur expérimenté préfère la solution suivante :

```
void copietab(char *S, char *T)
{
  while (*S++ = *T++);
}
```

La deuxième formulation de cette fonction est élégante, compacte, efficace et la traduction en langage machine fournit un code très rapide... ; mais bien que cette manière de résoudre les problèmes soit le cas normal en C, il n'est pas si évident de suivre le raisonnement.

Conclusions Bien entendu, dans les deux exemples ci-dessus, les formulations 'courtes' représentent le bon style dans C et sont de loin préférables aux deux autres. Nous constatons donc que :

— la programmation efficiente en C nécessite beaucoup d'expérience et n'est pas facilement accessible à des débutants.

— sans commentaires ou explications, les programmes peuvent devenir incompréhensibles, donc inutilisables.

(3) **portabilité et bibliothèques de fonctions :** La portabilité est l'un des avantages les plus importants de C : en écrivant des programmes qui respectent le standard ANSI-C, nous pouvons les utiliser sur n'importe quelle machine possédant un compilateur ANSI-C. D'autre part, le répertoire des fonctions ANSI-C est assez limité. Si un programmeur désire faire appel à une fonction spécifique de la machine (p.ex : utiliser une carte graphique spéciale), il est assisté par une foule de fonctions 'préfabriquées', mais il doit être conscient qu'il risque de perdre la portabilité. Ainsi, il devient évident que les avantages d'un programme portable doivent être payés par la restriction des moyens de programmation.

(3) **discipline de programmation** (Les dangers de C) : C est un langage près de la machine, donc dangereux et bien que C soit un langage de programmation structuré, il ne nous force pas à adopter un certain style de programmation (comme p.ex. Pascal). Dans un certain sens, tout est permis et la tentation de programmer du 'code spaghetti' est grande. (Même la commande 'goto', si redoutée par les puristes ne manque pas en C). Le programmeur a donc beaucoup de libertés, mais aussi des responsabilités : il doit veiller lui-même à adopter un style de programmation propre, solide et compréhensible.

Voilà un exemple amusant et illisible de program C qui calcul beaucoup de décimales de PI.

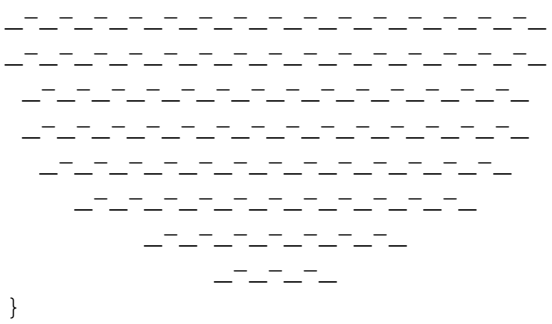
```
int a=10000,b,c=2800,d,e,f[2801],g;main(){for(;b-c;)f[b++]=a/5;for(;d=0,g=c*2;c
-=14,printf("%.4d",e+d/a),e=d%a)for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);}
```

la sortie qui est sur une ligne est

```
31415926535897932384626433832795028841971693993751
05820974944592307816406286208998628034825342117067
98214808651328230664709384460955058223172535940812
84811174502841027019385211055596446229489549303819
64428810975665933446128475648233786783165271201909
14564856692346034861045432664821339360726024914127
37245870066063155881748815209209628292540917153643
67892590360011330530548820466521384146951941511609
43305727036575959195309218611738193261179310511854
80744623799627495673518857527248912279381830119491
29833673362440656643086021394946395224737190702179
86094370277053921717629317675238467481846766940513
20005681271452635608277857713427577896091736371787
21468440901224953430146549585371050792276892589235
42019956112129021960864034418159813626977477130996
05187072113499999983729780499510597317328160963185
```

Un autre programme qui calcul une aproximation de π avec un programme plus joli, mais tout aussi illisble :

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
```



```
}
```

Il est un site web `http://ww.ioccc.org/` pour le « The International Obfuscated C Code Contest »

3.2 Un exemple assez complet

Les programmes en `C` sont composés essentiellement de fonctions et de variables. Généralement le programme est formé de plusieurs fichiers. Des fichiers contenant les déclarations qui ont pour suffixe `.h` et des fichiers contenants les fonctions de suffixe `.c`. Tout programme `C` doit contenir la fonction `int main()` qui est le point d'entrée. Cette fonction retourne une valeur entière qui est nulle si tout est correct, non nulle sinon (il y a un erreur).

Voila un exemple qui montre beaucoup de point de la syntaxe du `C` , avec de la compilation séparer et un makefile pour compiler le tout. C'est un condensé de `C`.

Les trois fichier de l'exemple :

le fichier `lesson.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
/*  definition des structures */
#include <math.h>
#include "lesson.h"

int main(int argc,char ** argv)
{
 /*  les declarations */
    int i;
    float f;
    double d,Pi, *dt;
    long l;

    /*  tableau de taille fixe */
    int tabi[10];
    double tabd[10];
    /*     des pointeurs  <=> tableau de taille inconnue */
    int *p2i;     /*     pointeur sur un entier */
    double *p2d;    /*     pointeur sur un double  */
```

```
char  str[80];  /*    une chaîne de caractères est un tableau
  de caractères se terminant  par le caractère \0 */

list  * head, *p;  /* de pointeur sur un type list définit par un typedef dans
lesson.h  */


printf("Hello World\n");
/*      */
Pi = 4* atan(1);
l= 10L;
f=10.0;
i = (argc>=2 ?  atoi(argv[1]) : 100);
dt = allocation_dynamique(i);
/*     le taille de type de base */
printf(" Size of char %d, int %d , float %d, double %d , long %d "
       ",  long long %d\n",
       sizeof(char), sizeof(int),sizeof(float),
       sizeof(double),sizeof(long),sizeof(long long));
printf(" Size of   ,tabd %d, p2i %d, p2d %d \n",sizeof(tabd),
   sizeof(p2i),sizeof(p2d));
for (i=0;i<5;i=i+1)
  routine(i);
for (i= 0;i<10;i++)
  tabi[i]=i*i;
for (i= 0;i<10;i++)
  tabd[i]=cos(i*Pi);
/*     construction de la liste */
head=0; /*   liste vide */
for (i=1;i<=5;i++)
  head=NouvelleFeuille(i,head); /*    ajoute en tête de liste */

/*    parcours type 1  -------------------- */
for (p=head;p;p=p->next)
  printf("no %d  Value %d \n",p->n,p->value);
/*    parcours type 2  -------------------- */
 d=0;
 p=head;
 while(p)
  {
    d += p->value;
    p=p->next;
    if (d< 100) break; /*    pour sortir de la boucle */
  }
 printf(" Somme en double %lf\n",d);
/*                       -------------------- */
/*  --- un petit test  avec goto */
 if (d < 10)  goto L1;
   printf(" il est faux que  d < 10 \n");
 L1:
 /* --- initialisation du tableau de pointeur de fonction */

 initilisation_tab_func();

 d += tab_func[0](d,d); /* appelle à d'un fonction du tableau de fonction */

 printf(" d  = %f \n",d);
```

```
    /* verification des variables global static */
    file = "lesson.c";
    p_file_r("lesson.c");
    p_file_l("lesson.c");

    copy_chaine(str,"CouCou ?");
    printf(" '%s'\n",str);
    free(dt); /*    liberation de la memoire alloue  */

    /*     lecture formaté */
    printf(" entrez un entier et un double ? ");
    scanf("%d%lf",&i,&d);
    printf("\n i = %d  d= %lf\n",i,d);
    return 0;
}

 void p_file_l(char * ou)
 {
%    printf("p_file_l: dans %s \t file= %s \n",ou,file);
 }

void operateurs()
 {

   double a=1,b=2,c;
   long i=3,j=4,k=8;
   double *p, tab[10];
   c = (a*b) - (a/b) - 1 ;
   c = pow(a,10); /*     puissance */
   i=10;
   j=3;
   k= i%3; /*    reste de la division signé */
   /*
     expression de comparaison
       a < b , a > b,    a <= b,     a >= b
       a == b;     a != b   différent
     expression boolean:
        faux   <=> valeur nulle de tout type
        vraie  <=> valeur non nulle de tout type
       ! (non),
        && (et),  le membre de droit n'est pas évalué si membre de gauche est faux
        || (ou),  le membre de droit n'est pas évalué si membre de gauche est vrai
     operateur sur le cha{ı}ne de bits
        & (et) ,| (ou inclusif) ,^ (ou exclusif), ~ (complement (non))
        << (lshift), >> (lshift)

     les operateurs affectations
       =, += (ajoute à) , -= (ote à), /= (divise par) , *=, %=, ...

     les operateurs incrémentation et décrémentation
       ++, -- d'une variable
   */
    i=1;
    j=i++; /*     ici j==1 et i==2 */
    j=++i; /*     ici j==3 et j==3 */
```

```
    /*    attention à l'arithmétique des pointeurs,
       unité est la taille de l'objet pointé */
    p=tab;      /*    p pointe sur tab[0] */
    *p = 10.1; /*    défini la valeur pointe par le pointeur */
    ++p;        /*    pointe sur tab[1] */
    *++p=3;     /*      tab[2] = 2.1 <=> (*tab+2) = 2.0 */
    p=&c;       /*     &d  adresse de c <=> le pointeur sur c */
 }
void des_types ()
{
   int i;
   int *pi;                /*    pointeur sur un int */
   int i5[]                /*    un tableau de 5 int  initialiser*/
     = {0,1,2,3,4};
   int i2x5[2][5]           /*    un tableau de 2x5 int */
     ={ {11,12,13,14,15},
        {21,22,23,24,25}};
   int (*pi5)[5];           /*    un pointeur sur un tableau de 5 entier */
   int *ip5[5];             /*    un tableau de 5 pointeur sur des entiers */

   pi = & i;    /*    le pointeur pointe sur l'entier i */
   pi5= &i5;   /*    le pointeur pointe sur le tableau i5 */

   for (i=0;i<5;i++)
     ip5[i]= &(i2x5[1][5]); /*    initialisation du tableau de pointeur */

   i=1;
   *pi += 2; /*    *pi et i sont la même mémoire donc */
   assert( i==3); /*     génère une erreur à l'execution  si l'assertion est fausse*/
}

double * allocation_dynamique (int n)
{
  double *tab,Pi_n;
  int i;
  assert(n>0);
  Pi_n= 4*atan(1)/n;
  /*    allocation de n double en memoire de 0 a n-1 */
  tab = (double *) malloc(sizeof(double)*n);
  for (i=0;i<n;i++)
    tab[i]=sin(i*Pi_n);
  /*     il faudra détruire le tableau tab avec free(tab) */
  return tab;
}

/*    exemple de code pour copie une cha{ı}ne de caractères*/
void copy_chaine(char * dst,char * src)
{  while (*dst++=*src++); }
```

le fichier `lesson.h`

```
/*    dans le .h */
```

```
void routine(int i);

/*     definition d'un type  structure de liste en C   */
typedef  struct list {
  int n;                  /*    numero de la feuille */
  int value;              /*    un value  */
  struct  list * next;  /*    suivant */
  } list;

list *NouvelleFeuille(int v,list *n);

/*    declaration un tableau de pointeur de 2 functions
   double  (*) (double, double) extern*/
extern  double (*tab_func[2])(double,double);

 void initilisation_tab_func();
 /*    une global static est local a l'unité de compilation */
static char * file;

 void p_file_l(char * ou);
 void p_file_r(char * ou);

double * allocation_dynamique (int n);
void operateurs();
double * allocation_dynamique (int n);
void copy_chaine(char * dst,char * src);
void des_types ();
```

le fichier `routines.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "lesson.h"

/*     3 functions locales à l'unité de compilation routines.c
    car elle sont static  */
static double  Add(double x,double y) {
  return x+y;
}
static double  Mul(double x,double y) {
  return x*y;
}
/*    fin des routines static (locales à l'unité de compilation) */

/*     definition de la variable global tab_func */
double (*tab_func[2])(double,double);

 void initilisation_tab_func()
 {
   tab_func[0]=Add;
   tab_func[1]=Mul;
   file = "routines.c";
 }
void routine(int i)
```

```
 {
   if (i) printf("routine True %d \n",i);
   else   printf("routine false %d \n",i);

   switch (i) {
     case 0: printf(" 0,"); break;
     case 1: printf(" 1,");
     case 2: printf(" 2,"); break;
     case 3:
     case 4: printf(" 3 ou 4,"); break;
     default:  printf (" autre cas \n");
   }
 }

/*     construction d'une feuille de la liste */
list *NouvelleFeuille(int v,list *n)
{
  static int nb=0;
  list * l= (list *) malloc(sizeof(list));
  nb++;
  l->n=nb;
  l->value=v;
  l->next= n;
  return l;
}

 void p_file_r(char * ou)
 {
   printf("p_file_r: dans %s \t file= %s\n",ou,file);
   p_file_l("p_file_r");
 }
```

le fichier `Makefile` ou le symbole <tabulation> est le caractère de tabulation.

```
OBJ=lesson.o routines.o
all:  lesson  westley pi

lesson: $(OBJ)
<tabulation>$(CC) -o lesson $(OBJ) -lm
```

pour compile les 3 exemples `lesson westley pi`, taper la commande shell. `make`, ou pour compile seulement le programme « lesson »taper la commande shell. `make lesson`. le résultat des commandes

```
hecht@kiwi:/disc2/hecht/lecon_de_C$ ls
Makefile lesson lesson.c lesson.h pi.c routine.c routines.c westley.c
hecht@kiwi:/disc2/hecht/lecon_de_C$ make
cc    -c -o lesson.o lesson.c
cc    -c -o routines.o routines.c
cc -o lesson lesson.o routines.o -lm
cc     westley.c   -o westley
cc     pi.c   -o pi
hecht@kiwi:/disc2/hecht/lecon_de_C$
```

et l'execution de la commande donne :

```
hecht@kiwi:/disc2/hecht/lecon_de_C$ ./lesson
Hello World
 Size of char 1, int 4 , float 4, double 8 , long 4 , long long 8
 Size of   ,tabd 80, p2i 4, p2d 4
routine false 0
 0,routine True 1
 1, 2,routine True 2
 2,routine True 3
 3 ou 4,routine True 4
 3 ou 4,n'27 5  Value 5
n'27 4  Value 4
n'27 3  Value 3
n'27 2  Value 2
n'27 1  Value 1
 Somme en double 5.000000
 d  = 15.000000
p_file_r: dans lesson.c     file= routines.c
p_file_l: dans p_file_r     file= lesson.c
p_file_l: dans lesson.c     file= lesson.c
 'CouCou ?'
 entrez un entier et un double  ? 123 123.02

 i = 123  d= 123.020000
hecht@kiwi:/disc2/hecht/lecon_de_C$
```

4 C++, quelques éléments de syntaxe

Il y a tellement de livres sur la syntaxe du C++ qu'il me paraît déraisonnable de réécrire un chapitre sur ce sujet, je vous propose le livre de Thomas Lachand-Robert qui est disponible sur la toile à l'adresse suivante http://www.ann.jussieu.fr/courscpp/, ou le cours C,C++ plus moderne aussi disponible sur la toile http://casteyde.christian.free.fr/cpp/cours. Pour avoir une liste à jour lire la page http://www.developpez.com/c/cours/. Bien sur, vous pouvez utiliser le livre The C++ , programming language [Stroustrup-1997]

Je veux décrire seulement quelques trucs et astuces qui sont généralement utiles comme les déclarations des types de bases et l'algèbre de typage.

Donc à partir de ce moment je suppose que vous connaissez, quelques rudiments de la syntaxe C++ . Ces rudiments que je sais difficile, sont (pour les connaître, il suffit de comprendre ce qui est écrit après) :

— le lexique infomatique à connaitre par coeur :

Mémoire
Octet
Type
Adresse
Ecriture mémoire
Lecture mémoire
Affectation
Fonction
Paramètre
Pointeur
Référence
Argument
Variable
Polymorphisme
Surcharge d'opérateur
Méthode
Objet
Classe
Instance
Opérateur
Conversion
Sémantique
Mémoire cache
Mémoire secondaire
Terminal
Bibliothèque
Edition de lien
Compilation
Répertoire
Make
Editeur de texte
Operateur de copie
Affectation par copie
Conversion
Evaluation
Execution
Executable
Shell script
Processus
Entre-Sortie
Fichier
Dynamique
Statique
Iteration
Boucle
Recursivité
Implementation
Généricité
Hérédité
Algorithme
Pile
Debugger
Compilateur
Préprocesseur
Complexité
url

Exemple de phase à comprendre : Une variable locale et dynamique d'une fonction n'existe en mémoire que pendant l'exécution la fonction, si la fonction est récursive cette fonction peux être plusieurs fois en mémoire comme cette variable, lorsqu'une variable locale statique d'une fonction existe toujours pendant le temps d'exécution et est unique.

— expression gauche
— expression droite
— Les types de base, les définitions de pointeur et référence (je vous rappelle qu'une référence est défini comme une variable dont l'adresse mémoire est connue et cet adresse n'est pas modifiable, donc une référence peut être vue comme une pointeur constant automatiquement déférencé, ou encore comme « donné un autre nom à une zone mémoire de la machine »).
— L'écriture d'un fonction, d'un prototypage,
— Les structures de contrôle associée aux mots clefs suivants : `if`, `else`, `switch`, `case`, `default`, `while`, `for`, `repeat`, `continue`, `break`.
— L'écriture d'une classe avec constructeur et destructeur, et des méthodes membres.
— Les passages d'arguments
  — par valeur (type de l'argument sans `&`), donc une copie de l'argument est passée à la fonction. Cette copie est créée avec le constructeur par copie, puis est détruite avec le destructeur. L'argument ne peut être modifié dans ce cas.
  — par référence (type de l'argument avec `&`) donc pas l'utilisation du constructeur par copie.
  — par référence sur un expression droite (type de l'argument avec `&&`), cet argument va être détruit sous peu donc on peut faire un échange.
  — par pointeur (le pointeur est passé par valeur), l'argument peut-être modifié.
  — paramètre non modifiable (cf. mot clef `const`).
  — La valeur retournée par copie (type de retour sans `&`) ou par référence (type de retour avec `&`)
— Polymorphisme et surcharge des opérateurs. L'appel d'une fonction est déterminée par son nom et par le type de ses arguments, il est donc possible de créer des fonctions de même nom pour des type différents. Les opérateurs n-naire (unaire `n=1` ou binaire `n=2`) sont des fonctions à `n` argument de nom `operator` ♣ `(n-args )` où ♣ est l'un des opérateurs du `C++` :

```
+    -    *    /    %    ^    &  |    ~    !    =    <       >       +=
-=   *=   /=   %=   ^=   &=   |= <<   >>   <<=  >>=  ==      !=      <=
>=   &&   ||   ++   --   ->*  , ->   []   ()   new  new[]   delete  delete[]
(T)
```

où (T est une expression de type), et où `(n-args )` est la déclaration classique des `n` arguments. Remarque si opérateur est défini dans une classe alors le premier argument est la classe elle même et donc le nombre d'arguments est $n-1$.

— Les règles de conversion (« cast » en Anglais) d'un type T en A par défaut qui sont génèrées à partir d'un constructeur `A(T)` dans la classe A ou avec l'opérateur de conversion `operator (A) ()` dans la classe T, `operator (A) (T)` hors d'une classe. De plus il ne faut pas oublier que `C++` fait automatiquement un au plus un niveau de conversion pour trouver la bonne fonction ou le bon opérateurs.
— Programmation générique de base (c.f. `template`). Exemple d'écriture de la fonction `min` générique suivante `template<class T> T & min(T & a,T & b){return a<b? a :b;}`

— Pour finir, connaître seulement l'existence du macro générateur et ne pas l'utiliser.

4.1 Les déclarations du C++

Les types de base du C++ sont respectivement : `bool`, `char`, `short`, `int`, `long`, `long long`, `float`, `double`, plus des pointeurs, ou des références sur ces types, des tableaux, des fonctions sur ces types et les constantes (objet non modifiable). Le tout nous donne une algèbre de type qui n'est pas triviale.

Voilà les principaux types généralement utilisé pour des types `T`,`U` :

déclaration	Prototypage	description du type en français
`T * a`	`T *`	un pointeur sur `T`
`T a[10]`	`T[10]`	un tableau de `T` composé de 10 variable de type T
`T a(U)`	`T a(U)`	une fonction qui a `U` retourne un `T`
`T &a`	`T &a`	une référence sur un objet de type `T`
`T &&a`	`T &a`	une référence sur une rvalue de type `T` qui ne sera jamais util
`const T a`	`const T`	un objet constant de type `T`
`T const * a`	`T const *`	un pointeur sur objet constant de type `T`
`T * const a`	`T * const`	un pointeur constant sur objet de type `T`
`T const * const a`	`T const * const`	un pointeur constant sur objet constant
`T * & a`	`T * &`	une référence sur un pointeur sur `T`
`T ** a`	`T **`	un pointeur sur un pointeur sur `T`
`T * a[10]`	`T *[10]`	un tableau de 10 pointeurs sur `T`
`T (* a)[10]`	`T (*)[10]`	un pointeur sur tableau de 10 `T`
`T (* a)(U)`	`T (*)(U)`	un pointeur sur une fonction `U` →`T`
`T (* a[])(U)`	`T (*[])(U)`	un tableau de pointeur sur des fonctions `U`→`T`
`T C:: *a`	`T C:: *`	pointeur sur une membre de type `T` dans la classe `C`
`T (C:: *a)(U)`	`T (C:: *)(U)`	pointeur sur une méthode membre `U` →`T` dans la classe `C`
...		

Remarque il n'est pas possible de construire un tableau de référence car il sera impossible à initialiser.

Exemple d'allocation d'un tableau data de *ldata* pointeurs de fonctions de R à valeur dans R :

```
R (**data)(R) = new (R (*[ldata])(R));
```

ou encore avec déclaration et puis allocation :

```
R (**data)(R); data = new (R (*[ldata])(R));
```

Pour l'utilisation des pointeurs sur des membres de classe `C`, (voir le point [Stroustrup-1997, C.12, page 853,] pour plus de details).

```
T (C:: *a)(U) = & C::func_T2U;
T C:: *a = & C:val_T;
```

Où `C` est une classe avec les deux membres `func_T2U` et `data_T`.

4.2 Comment Compile et éditer de liens

Comme en C ; dans les fichiers .cpp, il faut mettre les corps des fonctions et de les fichiers .hpp, il faut mettre les prototype, et la définition des classes, ainsi que les fonctions inline et les fonctions template, Voilà, un exemple complète avec trois fichiers a.hpp,a.cpp,tt.hpp, et un Makefile dans http://www.ann.jussieu.fr/~hecht/ftp/cpp/l1/a.tar.gz .

Remarque, pour déarchiver un fichier xxx.tar.gz, il suffit d'entrer dans une fenêtre shell tar zxvf xxx.tar.gz.

Listing 1: *(a.hpp)*

```
class A { public:
  A();                                          //    constructeur de la class A
};
```

Listing 2: *(a.cpp)*

```
#include <iostream>
#include "a.hpp"
using namespace std;

A::A()
{
  cout << " Constructeur A par défaut " << this << endl;
}
```

Listing 3: *(tt.cpp)*

```
#include <iostream>
#include "a.hpp"
using namespace std;
int main(int argc,char ** argv)
{
  for(int i=0;i<argc;++i)
    cout << " arg " << i << " = " << argv[i] << endl;
  A a[10];                                                    //    un tableau de 10 A
  return 0;                                                                  //    ok
}
```

les deux compilations et l'édition de liens qui génère un executable tt dans une fenêtre Terminal, avec des commandes de type shell; sh, bash tcsh, ksh, zsh, ... , sont obtenues avec les trois lignes :

```
[brochet:P6/DEA/sfemGC] hecht% g++ -c a.cpp
[brochet:P6/DEA/sfemGC] hecht% g++ -c tt.cpp
[brochet:P6/DEA/sfemGC] hecht% g++ a.o tt.o -o tt
```

Pour faire les trois choses en même temps, entrez :

```
[brochet:P6/DEA/sfemGC] hecht% g++  a.cpp tt.cpp -o tt
```

Puis, pour executer la commande `tt`, entrez par exemple :

```
[brochet:P6/DEA/sfemGC] hecht% ./tt aaa bb cc dd qsklhsqkfhsd " -----------
 arg 0 = ./tt
 arg 1 = aaa
 arg 2 = bb
 arg 3 = cc
 arg 4 = dd
 arg 5 = qsklhsqkfhsd
 arg 6 =  -----------
 Constructeur A par défaut 0xbfffeef0
 Constructeur A par défaut 0xbfffeef1
 Constructeur A par défaut 0xbfffeef2
 Constructeur A par défaut 0xbfffeef3
 Constructeur A par défaut 0xbfffeef4
 Constructeur A par défaut 0xbfffeef5
 Constructeur A par défaut 0xbfffeef6
 Constructeur A par défaut 0xbfffeef7
 Constructeur A par défaut 0xbfffeef8
 Constructeur A par défaut 0xbfffeef9
```

remarque : les `aaa bb cc dd qsklhsqkfhsd " ----------- "` après la commande `./tt` sont les paramètres de la commande est sont bien sur facultatif.

remarque, il est aussi possible de faire un `Makefile`, c'est à dire de créé le fichier :

Listing 4: *(Makefile)*

```
CXX=g++
CXXFLAGS= -g
LIB=
%.o:%.cpp
(caractere de tabulation  -->/)  $(CXX) -c $(CXXFLAGS) $^
tt: a.o tt.o
(caractere de tabulation  -->/)  $(CXX) tt.o  a.o -o tt
clean:
(caractere de tabulation  -->/) rm *.o tt

# les dependences
#
a.o: a.hpp  # il faut recompilé a.o si a.hpp change
tt.o: a.hpp  # il faut recompilé tt.o si a.hpp change
```

Pour l'utilisation :

- pour juste voir les commandes exécutées sans rien faire :
```
[brochet:P6/DEA/sfemGC] hecht% make -n tt
g++  -c  tt.cpp
g++  -c  a.cpp
g++  -o tt tt.o a.o
```
- pour vraiment compiler
```
[brochet:P6/DEA/sfemGC] hecht% make  tt
g++  -c  tt.cpp
g++  -c  a.cpp
g++  -o tt tt.o a.o
```
- pour recompiler avec une modification du fichier `a.hpp` via la command `touch` qui change la date de modification du fichier.
```
[brochet:P6/DEA/sfemGC] hecht% touch a.hpp
[brochet:P6/DEA/sfemGC] hecht% make tt
g++  -c  tt.cpp
g++  -c  a.cpp
g++  -o tt tt.o a.o
```
remarque : les deux fichiers sont bien à recompiler car il font un *include* du fichier `a.hpp`.
- pour nettoyer :
```
[brochet:P6/DEA/sfemGC] hecht% touch a.hpp
[brochet:P6/DEA/sfemGC] hecht% make clean
rm *.o tt
```

Remarque : Je vous conseille très vivement d'utiliser un `Makefile` pour compiler vous programme.

Exercice 3. Ecrire un Makefile pour compile et tester tous les programmes `http://www.ann.jussieu.fr/~hecht/ftp/cpp/l1/exemple_de_base`

4.3 Compréhension des constructeurs, destructeurs et des passages d'arguments

Exercice 4.

Faire une classe `T` avec un constructeur par copie et le destructeur, et la copie par affectation, et copie par déplacement : qui imprime quelque chose comme par exemple :

```
class T { public:
  T() { cout <<  "Constructeur par défaut " << this << "\n"}
  T(const  T & a) { cout <<"Constructeur par copie "
                         << this << "\n"}
  T(T && a) { cout <<"Constructeur par déplacement "
                         << this << " = " << &a <<  "\n"}

  ~T() { cout <<  "destructeur "<< this << "\n"}
  T & operator=(T & a) {cout << " copie par affectation :"
                         << this << " = " << &a << endl;}
  };
```

Puis tester, cette classe faisant un programme qui appelle les 4 fonctions suivantes, et qui contient une variable globale de type `T`.

```
T f1(T a){ return a;}
T f2(T &a){ return a;}
T &f3(T a){ return a;}            //    il y a un bug, le quel?
T &f4(T &a){ return a;}
T & f4(T &&a){ return a;}         //    il y a un bug, le quel?
```

Analysé et discuté très finement les résultat obtenus, et comprendre pourquoi, la classe suivant ne fonctionne pas, ou les opérateur programme font la même chose que les opérateurs par défaut.

```
class T { public:
  int * p;                                        //    un pointeur
  T() {p=new int;
       cout <<  "Constructeur par defaut " << this
            << " p=" << p << "\n"}
  T(const  T & a) {
       p=a.p;
       cout <<  "Constructeur par copie "<< this
            << " p=" << p << "\n"}
  T(T && a) {
       p=a.p;
       a.p=0;
       cout <<  "Constructeur par move "<< this
            << " p=" << p << "\n"}

  ~T() {
       cout <<  "destructeur "<< this
            << " p=" << p << "\n";
       delete p;}
  T & operator=(T & a) {
     cout << "copie par affectation  "<< this
          << "old  p=" << p
          <<  "new p="<< a.p << "\n";
     delete p;
```

4.4 Quelques règles de programmation

Malheureusement, il est très facile de faire des erreurs de programmation, la syntaxe du C++ n'est pas toujours simple à comprendre et comme l'expressibilité du langage est très grande, les possibilités d'erreur sont innombrables. Mais avec un peu de rigueur, il est possible d'en éviter un grand nombre.

La plupart des erreurs sont dû à des problèmes des pointeurs (débordement de tableau, destruction multiple, oublie de destruction), retour de pointeur sur des variable locales.

Voilà quelques règles à respecté.

Règle 1. *absolue* Dans une classe avec des pointeurs et avec un destructeur, il faut que les deux opérateurs de copie (création et affection) soient définis. Si vous considérez que ces deux opérateurs ne doivent pas exister alors les déclarez en privé sans les définir.

```
class sans_copie { public:
   long * p;                                                    //    un pointeur
   . . .
   sans_copie();
   ~sans_copie() { delete p;}
   private:
   sans_copie(const sans_copie &);            //    pas de constructeur par copie
   void operator=(const sans_copie &);           //    pas d'affection par copie
};
```

Dans ce cas les deux opérateurs de copies ne sont pas programmer pour qu'une erreur à l'édition des liens soit généré.

```
class avec_copie { public:
   long * p;                                                    //    un pointeur
   ~avec_copie() { delete p;}
   . . .
   avec_copie();
   avec_copie(const avec_copie &);          //    construction par copie possible
   void operator=(const avec_copie &);        //    affection par copie possible
};
```

Par contre dans ce cas, il faut programmer les deux opérateurs construction et affectation par copie.

Effectivement, si vous ne définissez ses opérateurs, il suffit d'oublier une esperluette (&) dans un passage argument pour que plus rien ne marche, comme dans l'exemple suivante :

```
  class Bug{ public:
   long * p;                                                    //    un pointeur
   Bug() p(new long[10]);
   ~Bug() { delete p;}
  };
  long & GetPb(Bug   a,int i){ return a.p[i];}                    //    copie puis
                                                          //    destruction de la copie
```

```
long & GetOk(Bug & a,int i){ return a.p[i];}                    //    ok

int main(int argc,char ** argv) {
 bug a;
 GetPb(a,1) = 1;                          //    bug le pointeur a.p est détruit ici
                                            //    l'argument est copie puis détruit
 cout << GetOk(a,1) << "\n";          //    bug on utilise un zone mémoire libérée

 return 0;                                //    le pointeur a.p est encore détruit ici
}
```

Le pire est que ce programme marche sur la plupart des ordinateurs et donne le résultat jusqu'au jour où l'on ajoute du code entre les 2 get (2 ou 3 ans après), c'est terrible mais ça marchait !...

Règle 2. Dans une fonction, ne jamais retournez de référence ou le pointeur sur une variable locale

Effectivement, retourner une référence sur une variable local implique que l'on retourne l'adresse mémoire de la pile, qui est libéré automatique en sortie de fonction, et qui est donc invalide hors de la fonction. mais bien sur le programme écrire peut marche avec de la chance.

Il ne faut jamais faire ceci :

```
int & Add(int i,int j)
{ int l=i+j;
  return l; }                              //    bug return d'une variable local l
```

Mais vous pouvez retourner une référence définie à partir des arguments, ou à partir de variables static ou global qui sont rémanentes.

Règle 3. Si, dans un programme, vous savez qu'un expression logique doit être vraie, alors vous devez mettre une assertion de cette expression logique.

Ne pas penser au problème du temps calcul dans un premier temps, il est possible de retirer toutes les assertions en compilant avec l'option `-DNDEBUG`, ou en définissant la macro du preprocesseur `#define NDEBUG`, si vous voulez faire du filtrage avec des assertions, il suffit de définir les macros suivante dans un fichier `http://www.ann.jussieu.fr/~hecht/ftp/cpp/assertion.hpp` qui active les assertions

```
#ifndef ASSERTION_HPP_
#define ASSERTION_HPP_
//    to compile all assertion
//   #define ASSERTION
//    to remove all the assert
//   #define NDEBUG
#ifndef ASSERTION
#define ASSERTION(i)  0
#else
#include <cassert>
#undef ASSERTION
#define ASSERTION(i) assert(i)
#endif
#endif
```

comme cela il est possible de garder un niveau d'assertion avec `assert`. Pour des cas plus fondamentaux et qui sont négligeables en temps calcul. Il suffit de définir la macro `ASSERTION` pour que les testes soient effectués sinon le code n'est pas compilé et est remplacé par `0`.
Il est fondamental de vérifier les bornes de tableaux, ainsi que les autres bornes connues. Aujourd'hui je viens de trouver une erreur stupide, un déplacement de tableau dû à l'échange de 2 indices dans un tableau qui ralentissait très sensiblement mon logiciel (je n'avais respecté cette règle).
Exemple d'une petite classe qui modélise un tableau d'entier

```
class Itab{ public:
  int n;
  int *p;
  Itab(int nn)
   { n=nn);
     p=new int[n];
     assert(p);}                                    //    vérification du pointeur
  ~Itab()
   { assert(p);                                     //    vérification du pointeur
     delete p;
     p=0;}                                  //    pour éviter les doubles destruction
  int & operator[](int i) { assert( i >=0 && i < n && p ); return p[i];}

private:       //    la règle 1 : pas de copie par défaut il y a un destructeur
  Itab(const Itab &);                           //    pas de constructeur par copie
  void operator=(const Itab &);                 //    pas d'affection par copie
}
```

Règle 4. N'utilisez le macro générateur que si vous ne pouvez pas faire autrement, ou pour ajoutez du code de vérification ou test qui sera très utile lors de la mise au point.

Règle 5. Une fois toutes les erreurs de compilation et d'édition des liens corrigées, il faut éditer les liens en ajoutant `CheckPtr.o` (le purify du pauvre) à la liste des objets à éditer les liens, afin de faire les vérifications des allocations.

Corriger tous les erreurs de pointeurs bien sûr, et les erreurs assertions avec le débogueur.

4.5 Vérificateur d'allocation

L'idée est très simple, il suffit de surcharger les opérateurs `new` et `delete`, de stocker en mémoire tous les pointeurs alloués et de vérifier avant chaque déallacation s'il fut bien alloué (cf. `AllocExtern::MyNewOperator(size_t )` et `AllocExternData.MyDeleteOperator(void *)`.
Le tout est d'encapsuler dans une classe `AllocExtern` pour qu'il n'y est pas de conflit de nom. De plus, on utilise `malloc` et `free` du C, pour éviter des problèmes de récurrence infinie dans l'allocateur. Pour chaque allocation, avant et après le tableau, deux petites zones mémoire de 8 octets sont utilisées pour retrouver des débordement amont et aval.
Et le tout est initialisé et terminé sans modification du code source en utilisant la variable `AllocExternData` globale qui est construite puis détruite. À la destruction la liste des pointeurs non détruits est écrite dans un fichier, qui est relue à la construction, ce qui permet de déboguer les oublis de déallocation de pointeurs.

Remarque 1. *Ce code marche bien si l'on ne fait pas trop d'allocations, destructions dans le programme, car le nombre d'opérations pour vérifier la destruction d'un pointeur est en nombre de pointeurs alloués. L'algorithme est donc proportionnel au carré du nombre de pointeurs alloués par le programme. Il est possible d'améliorer l'algorithme en triant les pointeurs par adresse et en faisant une recherche dichotomique pour la destruction.*

Le source de ce vérificateur `CheckPtr.cpp` est disponible à l'adresse suivante `http://www.ann.jussieu.fr/~hecht/ftp/cpp/CheckPtr.cpp`. Pour l'utiliser, il suffit de compiler et d'éditer les liens avec les autres parties du programme.
Il est aussi possible de retrouver les pointeurs non désalloués, en utilisant votre débogueur favori (par exemple gdb).
Dans `CheckPtr.cpp`, il y a une macro du préprocesseur `DEBUGUNALLOC` qu'il faut définir, et qui active l'appel de la fonction `debugunalloc()` à la création de chaque pointeur non détruit. La liste des pointeurs non détruits est stockée dans le fichier `ListOfUnAllocPtr.bin`, et ce fichier est génèré par l'exécution de votre programme.
Donc pour déboguer votre programme, il suffit de faire :

1. Compilez `CheckPtr.cpp`.
2. Editez les liens de votre programme `C++` avec `CheckPtr.o`.
3. Exécutez votre programme avec un jeu de donné.
4. Réexécutez votre programme sous le debogueur et mettez un point d'arrêt dans la fonction `debugunalloc()` (deuxième ligne `CheckPtr.cpp` .
5. remontez dans la pile des fonctions appelées pour voir quel pointeur n'est pas désalloué.
6. etc...

Exercice 5. un Makefile pour compiler et tester tous les programmes du `http://www.ann.jussieu.fr/~hecht/ftp/cpp/l1/exemple_de_base` avec `CheckPtr`

4.6 valgrind, un logiciel de vérification mémoire

Pris de lla page `http://valgrind.org/`
Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.
The Valgrind distribution currently includes six production-quality tools : a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools : a heap/stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms : X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, ARM/Android (2.3.x), X86/Darwin and AMD64/Darwin (Mac OS X 10.6 and 10.7).
Utilisation compilez votre program `mymain` avec l'option $-g$

```
 MBA-de-FH:s5 hecht$ valgrind --dsymutil=yes ./Amain
==8133== Memcheck, a memory error detector
==8133== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==8133== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==8133== Command: ./Amain
```

```
==8133==
--8133-- run: /usr/bin/dsymutil "./Amain"
UNKNOWN task message [id 3229, to mach_task_self(), reply 0x2703]
UNKNOWN task message [id 3229, to mach_task_self(), reply 0x2703]
UNKNOWN task message [id 3414, to mach_task_self(), reply 0x2703]
--8133-- WARNING: unhandled syscall: unix:357
--8133-- You may be able to write your own handler.
--8133-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--8133-- Nevertheless we consider this a bug.  Please report
--8133-- it at http://valgrind.org/support/bug_reports.html.
 arg 0 = ./Amain
...
==8133==
  p = 0x100000000
==8133== Conditional jump or move depends on uninitialised value(s)
==8133==    at 0x1000011A1: main (Amain.cpp:20)
==8133==
==8133== Use of uninitialised value of size 8
==8133==    at 0x1000011A7: main (Amain.cpp:21)
==8133==
 *p = -17958193
  p = 0x7fff5fbff974
 *p = 10
  p = 0x7fff5fbff970
 *p = 20
  p = 0x101000040
 *p = 100
0
==8133==
==8133== HEAP SUMMARY:
==8133==     in use at exit: 0 bytes in 0 blocks
==8133==   total heap usage: 5 allocs, 5 frees, 3,204 bytes allocated
==8133==
==8133== All heap blocks were freed -- no leaks are possible
==8133==
==8133== For counts of detected and suppressed errors, rerun with: -v
==8133== Use --track-origins=yes to see where uninitialised values come from
==8133== ERROR SUMMARY: 21 errors from 5 contexts (suppressed: 0 from 0)
MBA-de-FH:s5 hecht$ 0
```

4.7 Le déboggeur en 5 minutes

Premièrement, le déboggeur est un programme `gdb` ou `ddd`, qui permet d'ausculter votre programme, il vous permettra peut être de retrouver des erreurs de programmations qui génèrent des erreurs à l'exécution.
Pour utiliser le déboggeur il faut premièrement compiler les programmes avec l'option de compilation `-g`, pour cela il suffit de l'ajouter à la variable `CXXFLAGS` dans le `Makefile`.

Voilà la liste des commandes les plus utiles à mon avis :

- **`run`** lance ou relance le programme à débogger, (ajouter les paramètres de la commande après `run`.
- **`c`** continue l'exécution
- **`s`** fait un pas d'une instruction (en entant dans les fonctions)
- **`n`** fait un pas d'une instruction (sans entrer dans les fonctions)
- **`finish`** continue l'exécution jusqu'à la sortie de la fonction (`return`)
- **`u`** sort de la boucle courante

p print la valeur d'une variable, expression ou tableau : `p x` ou `p *v@100` (affiche les 100 valeur du tableau défini par le pointeur `v`).

where montre la pile des appels

up monte dans la pile des appels

down descend dans la pile des appels

l listing de la fonction courante

l 10 listing à partir de la ligne 10.

info functions affiche toutes les fonctions connues, et `info functions tyty` n'affiche que les fonctions dont le nom qui contient la chaîne `tyty`,

info variables même chose mais pour les variables.

b main.cpp:100 défini un point d'arrêt en ligne 100 du fichier `main.cpp`

b zzz défini un point d'arrêt à l'entrée de la fonction `zzz`

b A::A() défini un point d'arrêt à l'entrée du constructeur par défaut de la classe A.

d 5 détruit le 5^{e} point d'arrêt.

watch défini une variable à tracer, le programme va s'arrêter quand cette variable va changer.

help pour avoir plus d'information en anglais.

Une petite ruse bien pratique pour débogger, un programme avant la fin (`exit`). Pour cela ajouter une fonction `zzzz` et utiliser la fonction `atexit` du système. C'est à dire qui votre programme doit ressembler à :

```
#include <cstdlib>
void zzzz(){}                          //    fonction vide qui ne sert qu'à débogger
int main(int argc,char **argv)
{
  atexit(zzzz);                        //    pour que la fonction zzzz soit exécutée
                                       //    avant la sortie en exit
```

sous `gdb` ou `ddd` entrez

```
  b zzzz
```

pour ajouter un point d'arrêt à l'appel de la fonction `zzzz`.

Pour finir, voilà, un petit exemple d'utilisation :

```
brochet:~/work/Cours/InfoBase/l4 hecht$ gdb mainA2
```

```
GNU gdb 6.3.50-20050815 (Apple version gdb-563)
....  bla bla ...
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-apple-darwin"...Reading symbols
for shared libraries .... done
```
(gdb) b main
```
Breakpoint 1 at 0x2c79: file Amain.cpp, line 16.
```
(gdb) b zzzz()
```
Breakpoint 2 at 0x2c36: file Amain.cpp, line 11.
```
(gdb) b Amain.cpp:19
```
Breakpoint 3 at 0x2c8e: file Amain.cpp, line 19.
```
(gdb) run
```
Starting program: /Users/hecht/work/Cours/InfoBase/l4/mainA2
Reading symbols for shared libraries . done
Breakpoint 1, main () at Amain.cpp:16
16        atexit(zzzz);
```
(gdb) c
```
Continuing.
Breakpoint 3, main () at Amain.cpp:19
19        A a(n),b(n),c(n),d(n);
```
(gdb) s *on fait un step: on entre dans le constructeur*
```
A::A (this=0xbffff540, i=100000) at A2.hpp:11
11        A(int i) : n(i),v(new K[i]) { assert(v);} // constructeur
```
(gdb) finish *on sort du constructeur*
```
Run till exit from #0  A::A (this=0xbffff540, i=100000) at A2.hpp:11
0x00002ca0 in main () at Amain.cpp:19
19        A a(n),b(n),c(n),d(n);
```
(gdb) n *on fait un pas sans entrer dans les fonctions*
```
21        for(int i=0;i<n;++i)
```
(gdb) l *Affichage du source de la fonction autour du point courant*
```
16        atexit(zzzz);
17
18        int n=100000;
19        A a(n),b(n),c(n),d(n);
20        //  initialisation des tableaux a,b,c
21        for(int i=0;i<n;++i)
22          {
23            a[i]=cos(i);
24            b[i]=log(i);
25            c[i]=exp(i);
```
(gdb) *suite de l'Affichage du source*
```
26          }
27          d = (a+2.*b)+c*2.0;
28      }
```
(gdb) b 27 *defini un point d'arrêt en ligne 27 après la fin de boucle*
```
Breakpoint 4 at 0x2d50: file Amain.cpp, line 27.
```
(gdb) c *continue*
```
Continuing.
Breakpoint 4, main () at Amain.cpp:27
27          d = (a+2.*b)+c*2.0;
```

```
(gdb) d 4   supprime le point d'arrêt n°4
(gdb) p d   affiche la valeur de d
$1 = {
  n = 100000,
  v = 0x4c9008
}
(gdb) p d.v   affiche la valeur de d.v
$2 = (double *) 0x4c9008
(gdb) p *d.v@10     affiche les 10  valeurs pointées par d.v
$4 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) p *a.v@10     affiche les 10  valeurs pointées par a.v
$5 = {1, 0.54030230586813977, -0.41614683654714241, -0.98999249660044542,
 -0.65364362086361194, 0.28366218546322625, 0.96017028665036597,
  0.7539022543433046, -0.14550003380861354,
  -0.91113026188467694}
(gdb) p *b.v@10     affiche les 10  valeurs pointées par b.v
$6 = {-inf, 0, 0.69314718055994529, 1.0986122886681098, 1.3862943611198906,
1.6094379124341003, 1.791759469228055, 1.9459101490553132, 2.0794415416798357,
2.1972245773362196}     on remarquera que log(0) == -inf et ne génère par d'erreur
(gdb) p *c.v@10     affiche les 10  valeurs pointées par c.v
$7 = {1, 2.7182818284590451, 7.3890560989306504, 20.085536923187668,
54.598150033144236, 148.4131591025766, 403.42879349273511,
1096.6331584284585, 2980.9579870417283,  8103.0839275753842}
(gdb) c
Continuing.
Breakpoint 2, zzzz () at Amain.cpp:11
11        cout << " At exit " << endl;
(gdb) c
Continuing.
 At exit
                CheckPtr:Max Memory used   6250.000 kbytes  Memory undelete
0
Program exited normally.
(gdb) quit     On sort de gdb
```

5 Algorithmique

5.1 Introduction

Dans ce chapitre, nous allons décrire les notions d'algorithmique élémentaire, La notions de complexité.
Puis nous présenterons les chaînes et de chaînages ou liste d'abord d'un point de vue mathématique, puis nous montrerons par des exemples comment utiliser cette technique pour écrire des programmes très efficaces et simple.
Rappelons qu'une chaîne est un objet informatique composée d'une suite de maillons. Un maillon, quand il n'est pas le dernier de la chaîne, contient l'information permettant de trouver le maillon suivant. Comme application fondamentale de la notion de chaîne, nous commencerons par donner une méthode efficace de construction de l'image réciproque d'une fonction.
Ensuite, nous utiliserons cette technique pour construire l'ensemble des arêtes d'un maillage, pour trouver l'ensemble des triangles contenant un sommet donné, et enfin pour construire la structure creuse d'une matrice d'éléments finis.

5.2 Complexité algorithmique

Copié de http://fr.wikipedia.org/wiki/Complexité algorithmique
La théorie de la complexité algorithmique s'intéresse à l'estimation de l'efficacité des algorithmes. Elle s'attache à la question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?
Dans les années 1960 et au début des années 1970, alors qu'on en était à découvrir des algorithmes fondamentaux (tris tels que quicksort, arbres couvrants tels que les algorithmes de Kruskal ou de Prim), on ne mesurait pas leur efficacité. On se contentait de dire : « cet algorithme (de tri) se déroule en 6 secondes avec un tableau de 50 000 entiers choisis au hasard en entrée, sur un ordinateur IBM 360/91. Le langage de programmation PL/I a été utilisé avec les optimisations standard ». (cet exemple est imaginaire)
Une telle démarche rendait difficile la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé, etc.
Une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série The Art of Computer Programming. Il complétait cette analyse de considérations propres à la théorie de l'information : celle-ci par exemple, combinée à la formule de Stirling, montre qu'il ne sera pas possible d'effectuer un tri général (c'est-à-dire uniquement par comparaisons) de N élements en un temps croissant moins rapidement avec N que N ln N sur une machine algorithmique (à la différence peut-être d'un ordinateur quantique).

Pour qu'une analyse ne dépende pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur, il faut utiliser comme unité de comparaison des « opérations élémentaires » en fonction de la taille des données en entrée. Exemples d'opérations élémentaires : accès à une cellule mémoire, comparaison de valeurs, opérations arithmétiques (sur valeurs à codage de taille fixe), opérations sur des pointeurs. Il faut souvent préciser quelles sont les opérations élémentaires pertinentes pour le problème étudié : si les nombre manipulés restent de taille raisonnable, on considérera que l'addition de deux entiers prend un temps

constant, quels que soient les entiers considérés (ils seront en effet codés sur 32 bits). En revanche, lorsque l'on étudie des problèmes de calcul formel où la taille des nombres manipulés n'est pas bornée, le temps de calcul du produit de deux nombre dépendra de la taille de ces deux nombres.
On définit alors la taille de la donnée sur laquelle s'applique chaque problème par un entier lié au nombre d'éléments de la donnée. Par exemple, le nombre d'éléments dans un algorithme de tri, le nombre de sommets et d'arcs dans un graphe.
On évalue le nombre d'opérations élémentaires en fonction de la taille de la donnée : si 'n' est la taille, on calcule une fonction t(n).
Les critères d'analyse : le nombre d'opérations élémentaires peut varier substantiellement pour deux données de même taille. On retiendra deux critères :

- analyse au sens du plus mauvais cas : t(n) est le temps d'exécution du plus mauvais cas et le maximum sur toutes les données de taille n. Par exemple, le tri par insertion simple avec des entiers présents en ordre décroissants.
- analyse au sens de la moyenne : comme le « plus mauvais cas » peut en pratique n'apparaître que très rarement, on étudie tm(n), l'espérance sur l'ensemble des temps d'exécution, où chaque entrée a une certaine probabilité de se présenter. L'analyse mathématique de la complexité moyenne est souvent délicate. De plus, la signification de la distribution des probabilités par rapport à l'exécution réelle (sur un problème réel) est à considérer.

On étudie systématiquemnt la complexité asymptotique, noté grâce aux notations de Landau.

idée 1 : évaluer l'algorithme sur des données de grande taille. Par exemple, lorsque n est 'grand', $3n^3 + 2n^2$ est essentiellement $3n^3$.

idée 2 : on élimine les constantes multiplicatrices, car deux ordinateurs de puissances différentes diffèrent en temps d'exécution par une constante multiplicatrice. De $3 * n^3$, on ne retient que n^3

L'algorithme est dit en $0(n^3)$.
L'idée de base est donc qu'un algorithme en $0(n^a)$ est « meilleur » qu'un algorithme en $0(n^b)$ si $a < b$.
Les limites de cette théorie :
le coefficient multiplicateur est oublié : est-ce qu'en pratique $100 * n^2$ est « meilleur » que $5 * n^3$? l'occupation mémoire, les problèmes d'entrées/sorties sont occultés, dans les algorithmes numériques, la précision et la stabilité sont primordiaux. Point fort : c'est une notion indispensable pour le développement d'algorithmes efficaces.

Les principales classes de complexité :

- logarithmique : $\log_b n$
- linéaire : $an + b$
- polynomiale : $\sum_{i=0}^{n} a_i n^i$
- exponentielle : a^n
- factorielle : $n!$

Pour finir, il est aussi possible de parlé de la complexité mémoire, d'un algorithme.

5.3 Base, tableau, couleur

Recherche de la valeur maximale d'un tableau de `double u` de taille `N`.

```
double umax=u[0];
for (int i=1;i<N;++i)
  umax=max(umax,u[i]);
```

Si l'on veut connaitre l'indice `imax` associé à la plus grand valeur

```
int imax=0;
for (int i=1;i<N;++i)
  if(u[imax] < u[i])
    imax=i;
```

Exercice 6. Ecrire un programme trouve les 5 plus grande valeurs du tableau `u` avec une complexité de $O(N)$ en temps calcul et $O(1)$ en mémoire additionnelle.

5.3.1 Décalage d'un tableau

Nous voulons décaler le tableau de 1 comme suit formellement $u^{new}[i-1] = u^{old}[i], \forall i \in \{1, .., n-1\}$.

```
for (int i=1;i<N;++i)
  u[i-1]=u[i];
```

Remarque, tout ce passe bien car la valeur de $u[i-1]$ n'est plus utilisé dans la boucle, pour les i suivants mais dans le cas contraire $u^{new}[i+1] = u^{old}[i], \forall i \in \{0, .., n-2\}$, il suffit de faire la boucle à l'envers pour éviter le problème d'écrasement.

```
for (int i=N-1; i>1 ;--i )
  u[i]=u[i-1];
```

Ou si l'on ne veut pas changer le sens de parcours, alors il suffit de stocker les dépendances dans une deux mémoires auxiliaires en $u[i-1]$ et $u[i]$, ce qui donne

```
u0=u[0];                              //    valeur précédente avant modification
for (int i=1;i<N;++i)
  {
  u1=u[i];                                  //    valeur avant modification
  u[i]=u0;
  u0=u1;                              //    valeur précédente avant modification
  }
```

5.3.2 Renumérote un tableau

Soit `sigma` une permutation (bijection) de $\{0, ..., n-1\}$, le but est de changer l'ordre du tableau par la permutation σ ou par la permutation inverse $\sigma^{-}1$.
Il est difficile de faire ce type algorithme sur place, mais avec une copie, c'est trivail :

```
                                                    //    remumérotation
for (int i=0; i<n;++i )
  v[i]=u[sigma[i]];



                                                  //    remumérotation inverse
for (int i=0; i<n;++i )
  v[sigma[i]]=u[i];                          //    remarquons v[sigma[i]]=u[i];
```

Et donc pour construction la permutation inverse stocker dans le tableau `sigma1` il suffit d'ecrire :

```
                                                      //    permutation inverse
for (int i=0; i<n;++i )
  sigma1[sigma[i]]=i;
```

5.4 Construction de l'image réciproque d'une fonction

On va montrer comment construire l'image réciproque d'une fonction F. Pour simplifier l'exposé, nous supposerons que F est une fonction entière de $I = \{0, ..., n-1\}$ dans $J = \{0, ..., m-1\}$ et que ses valeurs sont stockées dans un tableau. Le lecteur pourra changer les bornes du domaine de définition ou de l'image sans grand problème.
Voici une méthode simple et efficace pour construire $F^{-1}(j)$ pour de nombreux j dans $\{0, ..., m-1\}$, quand n et m sont des entiers raisonnables. Pour chaque valeur $j \in Im\, F \subset \{0, ..., m-1\}$, nous allons construire la liste de ses antécédents. Pour cela nous utiliserons deux tableaux : `int head_F[m]` contenant les "têtes de listes" et `int next_F[n]` contenant la liste des éléments des $F^{-1}(i)$. Plus précisément, si i_1, i_2,...,$i_p \in [0, n]$, avec $p \geq 1$, sont les antécédents de j, `head_F[j]`$=i_p$, `next_F[`i_p`]`$=i_{p-1}$, `next_F[`i_{p-1}`]`$=i_{p-2}$, ..., `next_F[`i_2`]`$=i_1$ et `next_F[`i_1`]`$=$ -1 (pour terminer la chaîne).
L'algorithme est découpé en deux parties : l'une décrivant la construction des tableaux `next_F` et `head_F`, l'autre décrivant la manière de parcourir la liste des antécédents.

Algorithme 1. **Construction de l'image réciproque d'un tableau**

1. Construction :

```
int Not_In_I = -1;
 for (int j=0;j<m;j++)
  head_F[j]= Not_In_I;              //    initialement, les listes
                                    //    des antécédents sont vides
 for (int i=0;i<n;i++)
   {j=F[i];next_F[i]=head_F[j];head_F[j]=i;}  //   chaînage amont
```

2. Parcours de l'image réciproque de j dans $[0, n]$:

```
 for (int i=head_F[j];i!= Not_In_I;i=next_F[i])
  { assert(F[i]==j);          //    j doit être dans l'image de i
                                //    ... votre code
  }
```

Exercice 7. *Le pourquoi est laissé en exercice.*

5.5 Construction de classe d'equivalence

Le problème est très simple, nous avons une relation équivalence sur un ensemble d'entiers $I = \{0, \cdots, n-1\}$, qui n'ai défini que par ensemble de pair m d'entier noté a et qui est prolongé par transitivité. Pour compter le nombre de classe d'équivalence, où le nombre de composante connexe du graphe associé, un algorithme naturelle et trivial est en $n \times m$ et codé comme suit

```
int nbClassEquivalenceBrute(int n,int m,int (* a)[2])
 {
    vector<int> c(n);
    int nc = n;                                    //    nombre de composante
    for(int i=0;i<n;++i) ce[i]=i;
    for(int j=0, j<m; ++j)
      {
         int i1= a[j][0], i2=a[j][1];
         int c1=c[i1],c2=c[i2];      //    Attention il faut utiliser une copie
                                     //    car le tableau c va est modifié
         if(c1 !=c2)
          {
            nc--;
           for(int i= 0; i< n; ++i)
               if(c[i] == c2) c[i] = c1;
          }
      }
      return nc;
 }
```

Maintenant pour accélérer l'algorithme, nous allons associer un arbre à chaque composante et donc en chaque noeud on allons associer le père pour descendre dans l'arbre, et la racine qui n'a pas de père nous associons aucune valeur (code avec une valeur négative strict).

Donc pour trouver la racine r de l'arbre passant par i, il suffit d'écrire

```
int r = i;
while ( arbre[r] >=0) r=arbre[r];
```

Le coup calcul est majoré par la profondeur de l'arbre.
Pour fusionner les deux arbres de racine $r1$ et $r1$ et de profondeur respective $p1$ et $p2$, il suffit de écrire

- $arbre[r1] = r2$ et la racine commune sera $r2$ et la profondeur $max(p1 + 1, p2)$, ou bien
- $arbre[r2] = r1$ et la racine commune sera $r1$ et la profondeur $max(p1, p2 + 1)$,

si l'on stocke pour les racines une valeur négative donnant la profondeur de l'arbre associer, on va pouvoir fusionner les arbres pour obtenir l'arbre de profondeur minimal afin de limiter le coût calcul. Il est facile de montrer que la profondeur majore par $log_2(n)$.
Voici un façon simple de coder cette algorithme :

```
int nbClassEquivalence(int n,int m,int (* a)[2])
 {
    vector<int> arbre(n);
    int nc = n;                    //    nombre de composante = nombre de noeud
    for(int i=0;i<n;++i)
      arbre[i]=-1;  //    tous les arbre sont réduit à un noeud de profondeur 1

    for(int j=0, j<m; ++j)
      {
         int i1= a[j][0], i2=a[j][1], r1=i1,r2=i2,rr1,rr2;
         while ( (rr1=arbre[r1]) >=0) r1=rr1;
         while ( (rr2=arbre[r2]) >=0) r2=rr2;
         if(r1 !=r2)           //    racine différente, on fusionne les 2 arbres
          {
            nc--;                                    //    un composante de moins
            if( rr1<rr2) arbre[r2]=r1;             //    r1 plus profond : r2->r1
            else if( rr2<rr1) arbre[r1]=r2;        //    r2 plus profond : r1->r2
            else {arbre[r2]=r1;                      //    même profondeur r2->r1
                  arbre[r1]--;}          //   et la profondeur de r1 croît de 1
          }
      }
    return nc;
 }
```

De faite on a programmé l'algorithme de Kruskal `https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal` pour construire l'arbre de poids minimal si les pairs a ont été triés par coût croissant.

5.6 Tri par tas (heap sort)

La fonction `HeapSort` voir `http://fr.wikipedia.org/wiki/Tri_par_tas` pour les explications du trie par tas. La complexité de ce trie est $O(nlog_2(n))$.

```
template<class T>
void  HeapSort(T *c,int n) {
  c--;              //    because fortran version array begin at 1 in the routine
```

```
  int m,j,r,i;
  T crit;
  if( n <= 1) return;
  m = n/2 + 1;
  r = n;
  while (1) {
    if(m <= 1 ) {
      crit = c[r];
      c[r--] = c[1];
    if ( r == 1 ) { c[1]=crit; return;}
    } else  crit = c[--m];
    j=m;
    while (1) {
      i=j;
      j=2*j;
      if  (j>r) {c[i]=crit;break;}
      if ((j<r) && c[j] < c[j+1])) j++;
      if (crit < c[j]) c[i]=c[j];
      else {c[i]=crit;break;}
    }
  }
}
```

5.7 Construction des arêtes d'un maillage

Rappelons qu'un maillage est défini par la donnée d'une liste de points et d'une liste d'éléments (des triangles par exemple). Dans notre cas, le maillage triangulaire est implémenté dans la classe `Mesh` qui a deux membres `nv` et `nt` respectivement le nombre de sommets et le nombre de triangle, et qui a l'opérateur fonction `(int j,int i)` qui retourne le numero de du sommet `i` du triangle `j`. Cette classe `Mesh` pourrait être par exemple :

```
  class Mesh { public:
   int nv,nt;                                    //    nb de sommet, nb de triangle
   int (* nu)[3];                                                  //    connectivité
   double (* c)[2];                                              //    coordonnées de sommet
   int operator()(int i,int j) const { return nu[i][j];}
   Mesh(const char * filename);                                  //    lecture d'un maillage
 }
```

Ou bien sur la classe défini en 7.13.5.

Dans certaines applications, il peut être utile de construire la liste des *arêtes du maillage*, c'est-à-dire l'ensemble des arêtes de tous les éléments. La difficulté dans ce type de construction réside dans la manière d'éviter – ou d'éliminer – les doublons (le plus souvent une arête appartient à deux triangles).
Nous allons proposer deux algorithmes pour déterminer la liste des arêtes. Dans les deux cas, nous utiliserons le fait que les arêtes sont des segments de droite et sont donc définies complètement par la donnée des numéros de leurs deux sommets. On stockera donc les arêtes dans un tableau `arete[nbex][2]` où `nbex` est un majorant du nombre total d'arêtes. On pourra prendre grossièrement `nbex = 3*nt` ou bien utiliser la formule d'Euler en 2D

$$\texttt{nbe} = \texttt{nt} + \texttt{nv} + \texttt{nb\_de\_trous} - \texttt{nb\_composantes\_connexes}, \tag{1}$$

où nbe est le nombre d'arêtes (*edges* en anglais), nt le nombre de triangles et nv le nombre de sommets (*vertices* en anglais).
La première méthode est la plus simple : on compare les arêtes de chaque élément du maillage avec la liste de *toutes* les arêtes déjà répertoriées. Si l'arête était déjà connue on l'ignore, sinon on l'ajoute à la liste. Le nombre d'opérations est $nbe * (nbe + 1)/2$.
Avant de donner le première algorithme, indiquons qu'on utilisera souvent une petite routine qui échange deux paramètres :

```
template<class T> inline void Exchange (T& a,T& b) {T c=a;a=b;b=c;}
```

Algorithme 2. **Construction lente des arêtes d'un maillage $\mathscr{T}_h$**

```
int ConstructionArete(const Mesh & Th,int  (* arete)[2])
{
  int SommetDesAretes[3][2] = { {0,1},{1,2},{2,0}};
  nbe = 0;                                    //    nombre d'arête;
  for(int t=0;t<Th.nt;t++)
   for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]);
      int  j= Th(t,SommetDesAretes[et][1]);
      if (j < i) Exchange(i,j)            //    on oriente l'arête
      bool existe =false;      //    l'arête n'existe pas a priori
       for (int e=0;e<nbe;e++)    //   pour les arêtes existantes
        if (arete[e][0] == i && arete[e][1] == j)
          {existe=true;break;}  //    l'arête est déjà construite
      if (!existe)                              //    nouvelle arête
        arete[nbe][0]=i,arete[nbe++][1]=j;}
   return nbe;
}
```

Cet algorithme trivial est bien trop cher (en $O(9\,n^2)$) dès que le maillage a plus de 10^3 sommets (plus de 9 10^6 opérations). Pour le rendre de l'ordre du nombre d'arêtes, on va remplacer la boucle sur l'ensemble des arêtes construites par une boucle sur l'ensemble des arêtes ayant le même plus petit numéro de sommet. Dans un maillage raisonnable, le nombre d'arêtes incidentes sur un sommet est petit, disons de l'ordre de six, le gain est donc important : nous obtiendrons ainsi un algorithme en $9 \times nt$.
Pour mettre cette idée en oeuvre, nous allons utiliser l'algorithme de parcours de l'image réciproque de la fonction qui à une arête associe le plus petit numéro de ses sommets. Autrement dit, avec les notations de la section précédente, l'image par l'application F d'une arête sera le minimum des numéros de ses deux sommets. De plus, la construction et l'utilisation des listes, c'est-à-dire les étapes 1 et 2 de l'algorithme 3 seront simultanées.

Algorithme 3. **Construction rapide des arêtes d'un maillage $\mathscr{T}_h$**

```
int ConstructionArete(const Mesh & Th,int  (* arete)[2],
                      int nbex) {
   int SommetDesAretes[3][2] = { {1,2},{2,0},{0,1}};
   int end_list=-1;
   int * head_minv = new int [Th.nv];
   int * next_edge = new int [nbex];

   for ( int i =0;i<Th.nv;i++)
     head_minv[i]=end_list;                          //   liste vide

   nbe = 0;                                       //    nombre d'arête;

   for(int t=0;t<Th.nt;t++)
    for(int et=0;et<3;et++) {
      int  i= Th(t,SommetDesAretes[et][0]); //    premier sommet;
      int  j= Th(t,SommetDesAretes[et][1]);  //    second sommet;
      if (j < i) Exchange(i,j)            //    on oriente l'arête
      bool existe =false;      //    l'arête n'existe pas a priori
       for (int e=head_minv[i];e!=end_list;e = next_edge[e] )
                    //    on parcourt les arêtes déjà construites
        if ( arete[e][1] == j)  //    l'arête est déjà construite
          {existe=true;break;}                          //    stop
      if (!existe) {                           //    nouvelle arête
        assert(nbe < nbex);
        arete[nbe][0]=i,arete[nbe][1]=j;
                                       //    génération des chaînages
        next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;}
      }
  delete []  head_minv;
  delete []  next_edge;
  return nbe;
}
```

Preuve : la boucle `for(int e=head_minv[i];e!=end_list;e=next_edge[e])` permet de parcourir toutes des arêtes (i, j) orientées $(i < j)$ ayant même i, et la ligne :

```
next_edge[nbe]=head_minv[i],head_minv[i]=nbe++;
```

permet de chaîner en tête de liste des nouvelles arêtes. Le `nbe++` incrémente pour finir le nombre d'arêtes. ∎

Remarque : les sources sont disponible : `http://www.ann.jussieu.fr/~hecht/ftp/cpp/mesh-liste/BuildEdges.cpp`.

Exercice 8. *Il est possible de modifier l'algorithme précédent en supprimant le tableau* `next_edge` *et en stockant les chaînages dans* `arete[i][0]`*, mais à la fin, il faut faire une boucle de plus sur les sommets pour reconstruire* `arete[.][0]`*.*

Exercice 9. *Construire le tableau* `adj` *d'entier de taille* `3 nt` *qui donne pour l'arête* `p=i+3k` *(arête* `i` *du triangle* `k`*) , qui donne l'arête correspondante* `p'=i'+3k'` *.*

- *si cette arete est interne alors* `adj[i+3k]=i'+3k'` *où est l'arete* `i'` *du triangle* `k'`*, remarquons :* `i'=adj[i+3k]%3, k'=adj[i+3k]/3`*.*
- *sinon* `adj[i+3k]=-1`*.*

5.8 Construction des triangles contenant un sommet donné

La structure de données classique d'un maillage permet de connaître directement tous les sommets d'un triangle. En revanche, déterminer tous les triangles contenant un sommet n'est pas immédiat. Nous allons pour cela proposer un algorithme qui exploite à nouveau la notion de liste chaînée.
Rappelons que si `Th` est une instance de la class `Mesh` (voir 8.7), `i=Th(k,j)` est le numéro global du sommet `j`$\in [0,3[$ de l'élément `k`. L'application F qu'on va considérer associe à un couple (k,j) la valeur `i=Th(k,j)`. Ainsi, l'ensemble des numéros des triangles contenant un sommet `i` sera donné par les premières composantes des antécédents de `i`.
On va utiliser à nouveau l'algorithme 3, mais il y a une petite difficulté par rapport à la section précédente : les éléments du domaine de définition de F sont des *couples* et non plus simplement des entiers. Pour résoudre ce problème, remarquons qu'on peut associer de manière unique au couple `(k,j)`, où $j \in [0,m[$, l'entier `p(k,j)=k*m+j` [1]. Pour retrouver le couple `(k,j)` à partir de l'entier `p`, il suffit d'écrire que `k` et `j` sont respectivement le quotient et le reste de la division euclidienne de `p` par `m`, autrement dit :

$$\mathtt{p} \longrightarrow (\mathtt{k},\mathtt{j}) = (\mathtt{k} = \mathtt{p}/\mathtt{m}, \mathtt{j} = \mathtt{p}\%\mathtt{m}). \tag{2}$$

Voici donc l'algorithme pour construire l'ensemble des triangles ayant un sommet en commun :

Algorithme 4. **Construction de l'ensemble des triangles ayant un sommet commun**

Préparation :

```
int end_list=-1,
int *head_s = new int[Th.nv];
int *next_p = new int[Th.nt*3];
int i,j,k,p;
for (i=0;i<Th.nv;i++)
  head_s[i] = end_list;
for (k=0;k<Th.nt;k++)                          //    forall triangles
  for(j=0;j<3;j++) {
    p = 3*k+j;
    i = Th(k,j);
    next_p[p]=head_s[i];
    head_s[i]= p;}
```

Utilisation : parcours de tous les triangles ayant le sommet numéro `i`

```
for (int p=head_s[i]; p!=end_list; p=next_p[p])
{
   k=p/3;
   j = p % 3;
   assert( i == Th(k,j));
                                                //    votre code
}
```

1. Noter au passage que c'est ainsi que `C++` traite les tableaux à double entrée : un tableau `T[n][m]` est stocké comme un tableau à simple entrée de taille `n*m` dans lequel l'élément `T[k][j]` est repéré par l'indice `p(k,j) = k*m+j`.

Exercice 10. *Optimiser le code en initialisant* `p =-1` *et en remplaçant* `p = 3*j+k` *par* `p++`.

Remarque : les sources sont disponible : `http://www.ann.jussieu.fr/~hecht/ftp/cpp/mesh-liste/BuildListeTrianglesVertex.cpp`.

Remarque : en utilisant la STL, il est facile de construire un programme répondant a la question. La Construction en $O(nt\,log(nt))$ opération

```
vector<vector<int> > lst(Th.nv);
for (int k=0;k<Th.nt;++k)
  for(int j=0;j<3;++j)
    lst[Th(k,j)].push_back(k);
```

Utilisation pour parcours optimal de tous les triangles ayant le sommet numéro `i`

```
for (int l=0;l<lst[i].size();++l)
  {
   int k= lst[i][l];
   assert( Th(k,0) == i || Th(k,1) == i || Th(k,2) == i );
   ...
  }
```

5.9 Construction de la structure d'une matrice morse

Il est bien connu que la méthode des éléments finis conduit à des systèmes linéaires associés à des matrices très *creuses*, c'est-à-dire contenant un grand nombre de termes nuls. Dès que le maillage est donné, on peut construire le graphe des coefficients *a priori* non nuls de la matrice. En ne stockant que ces termes, on pourra réduire au maximum l'occupation en mémoire et optimiser les produits matrices/vecteurs.

5.9.1 Description de la structure morse

La structure de données que nous allons utiliser pour décrire la matrice creuse est souvent appelée "matrice morse" (en particulier dans la bibliothèque MODULEF), dans la littérature anglo-saxonne on trouve parfois l'expression "Compressed Row Sparse matrix" (cf. SIAM book...). Notons `n` le nombre de lignes et de colonnes de la matrice, et `nbcoef` le nombre de coefficients non nuls *a priori*. Trois tableaux sont utilisés : `a[k]` qui contient la valeur du k-ième coefficient non nul avec $k \in [0, nbcoef[$, `ligne[i]` qui contient l'indice dans `a` du premier terme de la ligne `i+1` de la matrice avec $i \in [-1, n[$ et enfin `colonne[k]` qui contient l'indice de la colonne du coefficient `k`$\in$`[0:nbcoef[`. On va de plus supposer ici que la matrice est symétrique, on ne stockera donc que sa partie triangulaire inférieure. En résumé, on a :

$$\mathtt{a[k]} = a_{\mathtt{ij}} \quad \text{pour } \mathtt{k} \in [\mathtt{ligne[i-1]}+1, \mathtt{ligne[i]}] \quad \text{et } \mathtt{j} = \mathtt{colonne[k]} \quad \text{si } \mathtt{i} \leq \mathtt{j}$$

et s'il n'existe pas de k pour un couple `(i,j)` ou si `i` $\leq$ `j` alors $a_{\mathtt{ij}} = 0$.
La classe décrivant une telle structure est :

```
class MatriceMorseSymetrique {
 int n,nbcoef;  //    dimension de la matrice et nombre de coefficients non nuls
```

```
  int *ligne,* colonne;
  double *a;
  MatriceMorseSymetrique(Maillage & Th);                    //    constructeur
  double*  pij(int i,int j) const;      //    retourne le pointeur sur le coef i,j
                                          //    de la matrice si il existe
}
```

Exemple : on considère la partie triangulaire inférieure de la matrice d'ordre 10 suivante (les valeurs sont les rangs dans le stockage et non les coefficients de la matrice) :

$$\begin{pmatrix} 0 & . & . & . & . & . & . & . & . & . \\ . & 1 & . & . & . & . & . & . & . & . \\ . & 2 & 3 & . & . & . & . & . & . & . \\ . & 4 & 5 & 6 & . & . & . & . & . & . \\ . & . & 7 & . & 8 & . & . & . & . & . \\ . & . & . & 9 & 10 & 11 & . & . & . & . \\ . & . & 12 & . & 13 & . & 14 & . & . & . \\ . & . & . & . & . & 15 & 16 & 17 & . & . \\ . & . & . & . & 18 & . & . & . & 19 & . \\ . & . & . & . & . & . & . & . & . & 20 \end{pmatrix}$$

On numérote les lignes et les colonnes de `[0..9]`. On a alors :

```
n=10,nbcoef=20,
ligne[-1:9] = {-1,0,1,3,6,8,11,14,17,19,20};
colonne[21] ={0, 1,  1,2,  1,2,3,  2,4,  3,4,5,  2,4,6,
              5,6,7,  4,8,  9};
a[21]                          //    ... valeurs des 21 coefficients de la matrice
```

5.9.2 Construction de la structure morse par coloriage

Nous allons maintenant construire la structure morse d'une matrice symétrique à partir de la donnée d'un maillage d'éléments finis P_1.
Les coefficient a_{ij} non nulles de la matrice sont tels qu'il existe un triangle K contenant les sommets i et j.
Donc, pour construire la ligne i de la matrice, il faut trouver tous les sommets j tels que i, j appartiennent à un même triangle. Ainsi, pour un noeud donné i, il s'agit de lister les sommets appartenant aux triangles contenant i. Le premier ingrédient de la méthode sera donc d'utiliser l'algorithme 6 pour parcourir l'ensemble des triangles contenant i. Mais il reste une difficulté : il faut éviter les doublons. Nous allons pour cela utiliser une autre technique classique de programmation qui consiste à « colorier »les coefficients déjà répertoriés : pour chaque sommet i (boucle externe), nous effectuons une boucle interne sur les triangles contenant i puis une balayage des sommets j de ces triangles en les coloriant pour éviter de compter plusieurs fois les coefficients a_{ij} correspondant. Si on n'utilise qu'une couleur, nous devons remettre la couleur du initial les sommets avant de passer à un autre i. Pour éviter cela, nous allons utiliser plusieurs couleurs, et on changera simplement de couleur de marquage à chaque fois qu'on changera de sommet i dans la boucle externe car toutes couleurs utilisées ont un numéro strictement inférieur.

Algorithme 5. **Construction de la structure d'une matrice morse**

```
MatriceMorseSymetrique::MatriceMorseSymetrique(const Mesh & Th){
  int i,j,jt,k,p,t;
  n = Th.nv;                                    //    nombre de ligne
  int color=0, * mark;
  mark = new int [n];    //    pour stocker la couleur d'un sommet
                         //    initialisation du tableau de couleur
  for(j=0;j<Th.nv;j++)  mark[j]=color;
  color++;                                     //    on change la couleur
     //   construction optimisee de l'image réciproque: i=Th(k,j)
  int end_list=-1,*head_s,*next_t;
  head_s = new int [Th.nv];
  next_p = new int [Th.nt*3];
  int i,j,k,p=0;
  for (i=0;i<Th.nv;i++)
    head_s[i] = end_list;
  for (k=0;k<Th.nt;k++)
   for(j=0;j<3;j++)
     { next_p[p]=head_s[i=Th(k,j)]; head_s[i]=p++;}
//    1) calcul du nombre de coefficients non nuls
//    a priori de la matrice pour pourvoir faire les allocations
  nbcoef = 0;
  for(i=0; i<n; i++,color++,nbcoef++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
       for(jt=0; jt< 3; jt++ )
        if(i <= (j=Th(t,jt)) && mark[j] != color))  //   nouveau j
        { mark[j]=color; nbcoef++;}     //    => marquage + ajout
//    2) allocations mémoires
  ligne = new int [n+1];
  ligne++;                          //    car le tableau commence en -1;
  colonne = new int [ nbcoef];
  a = new double [nbcoef];
//    3) constructions des deux tableaux ligne et colonne
  ligne[-1] = -1;
  nbcoef =0;
  for(i=0; i<n; ligne[i++]=nbcoef, color++)
    for (p=head_s[i],t=p/3; p!=end_list; t=(p=next_p[p])/3)
       for(jt=0; jt< 3; jt++ )
        if ( i <= (j=Th(t,jt)))  && mark[j] != color))
                     //    nouveau coefficient => marquage + ajout
         mark[j]=color, colonne[nbcoef++]=j;
//    4) tri des lignes par index de colonne
  for(i=0; i<n; i++)
    HeapSort(colonne + ligne[i-1] + 1 ,ligne[i] - ligne[i-1]);
//    nettoyage memoire
  delete  [] head_s;
  delete  [] next_p;
  delete  [] mark;
}
```

Au passage, nous avons utilisé la fonction `HeapSort` qui implémente un petit algorithme de tri, présenté dans [Knuth-1975], qui a la propriété d'être toujours en $nlog_2 n$ (*cf.* 8.6 page 132). Noter que l'étape de tri n'est pas absolument nécessaire, mais le fait d'avoir des lignes triées par indice de colonne permet d'optimiser l'accés à un coefficient de la matrice dans la structure

creuse, en utilisant une recherche dichotomique comme suit :

```
inline double*  MatriceMorseSymetrique::pij(int i,int j) const
 {
   assert(i<this->n && j< this->n);
   if (j > i ) {  int k=i; i=j;  j=k;}                    //    echange i et j
   int i0= ligne[i];
   int i1= ligne[i+1]-1;
   while (i0<=i1)                                              //    dichotomie
    {
      int im=(i0+i1)/2;
      if (j< colonne[im]) i1=im-1;
      else if (j> colonne[im]) i0=im+1;
      else return a+im;
    }
   return 0;                                            //    le coef n'existe pas
 }
```

Remarque : Si vous avez tout compris dans ces algorithmes, vous pouvez vous attaquer à la plupart des problèmes de programmation.

5.9.3 Le constructeur de la classe `SparseMatrix`

En modifiant légèrement l'algorithme précédent on obtient le constructeur de la classe `SparseMatrix` suivant :

```
template<class R>
template<class Mesh>
SparseMatrix<R>::SparseMatrix(Mesh & Th)
  :
  VirtualMatrice<R>(Th.nv),n(Th.nv),m(Th.nv), nbcoef(0),
  i(0),j(0),a(0)
  {
    const int nve = Mesh::Element::nv;
    int end=-1;
    int nn= Th.nt*nve;
    int mm= Th.nv;
    KN<int> head(mm);
    KN<int> next(nn);
    KN<int> mark(mm);
    int color=0;
    mark=color;

    head = end;
    for (int p=0;p<nn;++p)
      {
        int s= Th(p/nve,p%nve);
        next[p]=head[s];
        head[s]=p;
      }
    nbcoef =0;
    n=mm;
    m=mm;
    int kk=0;
```

```
  for (int step=0;step<2;++step)                                    //    2 etapes
                                          //    une pour le calcul du nombre de coef
                                                  //    l'autre pour la construction
    {
      for (int ii=0;ii<mm;++ii)
        {
          color++;
          int ki=nbcoef;
          for (int p=head[ii];p!=end;p=next[p])
            {
              int k=p/nve;
              for (int l=0;l<nve;++l)
                {
                  int jj= Th(k,l);
                  if( mark[jj] != color)   //    un nouveau sommet de l'ensemble
                    if (step==1)
                      {
                        i[nbcoef]=ii;
                        j[nbcoef]=jj;
                        a[nbcoef++]=R();
                      }
                    else
                      nbcoef++;
                  mark[jj]=color;                 //    on colorie le sommet j;
                }
            }
          int ki1=nbcoef;
          if(step==1)
            HeapSort(j+ki,ki1-ki);
        }

      if(step==0)
        {
          cout << " Allocation des tableaux  " << nbcoef << endl;
          i= new int[nbcoef];
          j= new int[nbcoef];
          a= new R[nbcoef];
          kk=nbcoef;
          nbcoef=0;
        }


    }
}
```

6 Exemples

7 Exemples

7.1 Le Plan $\mathbb{R}^2$

Voici une modélisation de $\mathbb{R}^2$ disponible à `http://www.ann.jussieu.fr/~hecht/ftp/cpp/l1/R2.hpp` qui permet de faire des opérations vectorielles et qui définit le produit scalaire de deux points A et B, le produit scalaire sera défini par (A, B).

7.1.1 La classe R2

```
//    Définition de la class R2
//    sans compilation sépare toute les fonctions
//    sous défini dans ce R2.hpp avec des inline
                                                                                //
//    remarque la fonction abort est déclaré dans #include <cstdlib>

//    definition R (les nombres reals)
typedef double R;                                    //    définition de R (les réel)

//    The class R2
class R2 {

public:
  R x,y;                                                    //    declaration des membre
                                                           //    les 3 constructeurs ---
  R2 () :x(0.),y(0.) {}                  //    rappel : x(0), y(0) sont initialiser
   R2 (R a,R b):x(a),y(b)  {}                    //    via le constructeur de double
  R2 (const R2 & a,const R2 & b):x(b.x-a.x),y(b.y-a.y)  {}

  //    le constucteur par defaut est inutile
                                          //    R2 ( const R2 & a) :x(a.x),y(a.y)

  //    rappel les operator definis dans une class on un parametre
  //    caché qui est la class elle meme (*this)

  //    les opérateurs affectations
  //    operateur affection = est inutil car par défaut,il est correct
              //    R2 & operator=( const R2 & P) x = P.x;y = P.y;return *this;
                                          //    les autre opérateur d'affectations
  R2 &  operator+=(const R2 & P)  {x += P.x;y += P.y;return *this;}
  R2 &  operator-=(const R2 & P) {x -= P.x;y -= P.y;return *this;}
  R2 &  operator*=(R a)  {x *= a;y *= a;return *this;}
  R2 &  operator/=(R a)  {x /= a;y /= a;return *this;}
  //    les operateur binaire + - * , ^
  R2   operator+(const R2 & P)const  {return R2(x+P.x,y+P.y);}
  R2   operator-(const R2 & P)const {return R2(x-P.x,y-P.y);}
  R    operator,(const R2 & P)const {return  x*P.x+y*P.y;} //    produit scalaire
  R    operator^(const R2 & P)const {return  x*P.y-y*P.x;}      //    produit det
  R2   operator*(R c)const {return R2(x*c,y*c);}
  R2   operator/(R c)const {return R2(x/c,y/c);}
                                                          //    operateur unaire
  R2   operator-()const  {return R2(-x,-y);}
  R2   operator+()const  {return *this;}
                                                                //    un methode
```

```
  R2   perp() const {return R2(-y,x);}                    //    la perpendiculaire

  //    les operators tableau
  //    version qui peut modifie la class via l'adresse de x ou y
  R & operator[](int i) { if(i==0) return x;                //    donc pas const
                          else if (i==1) return y;
                          else {assert(0);exit(1);};}
  //    version qui retourne une reference const qui ne modifie pas la class
  const R & operator[](int i) const { if(i==0) return x;        //    donc  const
                                      else if (i==1) return y;
                                      else {assert(0);exit(1);};}

  //    les operateurs fonction
                  //    version qui peut modifie la class via l'adresse de x ou y
  R & operator()(int i) { if(i==1) return x;                //    donc pas const
                          else if (i==2) return y;
                          else {assert(0);abort();};}
       //    version qui retourne une reference const qui ne modifie pas la class
  const R & operator()(int i) const { if(i==1) return x;        //    donc  const
                                      else if (i==2) return y;
                                      else {assert(0);abort();};}

};                                                          //    fin de la class R2

//    la class dans la class
inline std::ostream& operator <<(std::ostream& f, const R2 & P )
       { f << P.x << ' ' << P.y  ; return f; }
inline std::istream& operator >>(std::istream& f,  R2 & P)
       { f >>  P.x >>  P.y ; return f; }

inline R2 operator*(R c,const R2 & P) {return P*c;}
inline R2 perp(const R2 & P) { return R2(-P.y,P.x); }
inline R2 Perp(const R2 & P) { return P.perp(); }    //    un autre fonction perp
```

Quelques remarques sur la syntaxe

- Dans une classe (`class`) par défaut, toutes les définitions sont dans la sphère privé, c'est-à-dire quelle ne sont utilisable que par les méthodes de la classe ou les amis (c.f. `friend`) . L'ajout de `public:` permet de dire que tout ce qui suit est dans la sphère publique donc utilisable par tous. Et la seule différence entre un objet défini par `struct` et par `class` est la sphère d'utilisation par défaut qui est respectivement publique (`pubic`) ou privée (`private`) .
- Les membres de la classe (les données) ici sous les champs `x` et `y`. Ces 2 champs seront construction à la création de la classe par les constructeurs du la classe. Les constructeurs sont des méthodes qui ont pour nom, le nom de la classe, ici `R2`. Dans le constructeur `R2 (R a,R b):x(a),y(b) {}` les objets sont initialise par les constructeurs des membres entre le `:` et `{` des membres de la classe qu'il faut mettre dans le même ordre que l'ordre d'apparition dans la classe. . Le constructeur ici ne fait initialiser les deux champs `x` et `y`, il n'y a pas de code dans le corps (zone entre les `{}`) du constructeur.
- Les méthodes (fonctions membres) dans une classe ou structure ont un paramètre caché qui est le pointeur sur l'objet de nom `this`. Donc, les opérateurs binaires dans une classe n'ont seulement qu'un paramètre. Le premier paramètre étant la classe et le second étant le paramètre fourni.

— Si un opérateur ou une methode d'une classe ne modifie pas la classe alors il est conseillé de dire au compilateur que cette fonction est « constante »en ajoutant le mots clef **const** après la définition des paramètres.
— Dans le cas d'un opérateur défini hors d'une classe le nombre de paramètres est donné par le type de l'opérateur uniare (`+ - * ! [] etc..` : 1 paramètre), binaire (`+ - * / | & || &&^ == <= >= < > etc..` 2 paramètres), n-aire (`()` : n paramètres).
— `ostream`, `istream` sont les deux types classique pour respectivement écrire et lire dans un fichier ou sur les entrées sorties standard. Ces types sont définis dans le fichier « iostream »inclue avec l'ordre `#include<iostream>` qui est mis en tête de fichier ;
— les deux opérateurs `<<` et `>>` sont les deux opérateurs qui généralement et respectivement écrivent ou lisent dans un type `ostream`, `istream`, ou `iostream`.
— Il y a bien autre façon d'écrire ces classes. Dans l'archive `http://www.ann.jussieu.fr/~hecht/ftp/cpp/R2.tgz` , vous trouverez trois autres méthodes d'écriture de cette petite classe.

v1) Avec compilation sépare où toutes les mèthodes et fonctions sont prototypées dans le fichier `R2-v1.hpp` et elles sont défini dans le fichier `R2-v1.cpp`

v2) Sans compilation sépare où toutes les méthodes et fonctions sont définies dans le fichier `R2-v2.hpp`.

v3) Version avec des champs privés qui interdisent utilisation direct de `x` et `y` hors de la classe, il faut passer par les méthodes `X()` et `Y()`

7.1.2 Utilisation de la classe R2

Cette classe modélise le plan $\mathbb{R}^2$, pour que les opérateurs classiques fonctionnent, c'est-à-dire : Un point P du plan définit via modélisé par ces 2 coordonnées x, y, et nous pouvons écrire des lignes suivantes par exemple :

```
R2 P(1.0,-0.5),Q(0,10);
R2 O,PQ(P,Q);                          //    le point O est initialiser à 0,0
R2 M=(P+Q)/2;                           //    espace vectoriel à droite
R2 A = 0.5*(P+Q);                       //    espace vectoriel à gauche
R  ps = (A,M);                          //    le produit scalaire de R2
R  pm = A^M;                                  //    le determinant de A,M
                                  //    l'aire du parallélogramme formé par A,M
R2 B = A.perp();                        //    B est la rotation de A par π/2
R  a= A.x + B.y;
A  = -B;
A += M;                                                 //    ie. A = A + M;
A -= B;                                                    //    ie. A = -A;
double abscisse= A.x;                                 //    la composante x de A
double ordonne = A.y;                                 //    la composante y de A
A.y= 0.5;                                          //    change la composante de A
A(1) =5;                                    //    change la 1 composante (x) de A
A(2) =2;                                    //    change la 2 composante (y) de A
A[0] =10;                                   //    change la 1 composante (x) de A
A[1] =100;                                  //    change la 2 composante (y) de A
cout <<  A;                                //    imprime sur la console A.x et A.x
cint >> A;                        //    vous devez entrer 2 double à la console
```

Exercice 11.

Le but de cette exercice de d'affiche graphiquement les bassins d'attraction de la méthode de Newton, pour la résolution du problème $z^p - 1 = 0$, où $p = 3, 4, ...$
Rappel : la méthode de Newton s'écrit :

$$\text{Soit } z_0 \in \mathbb{C}, \quad z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)};$$

Q 1 Faire une classe `C` qui modélise le corps $\mathbb{C}$ des nombres complexes ($z = a + i * b$), avec toutes les opérations algébriques.

Q 2 Ecrire une fonction `C Newton(C z0,C (*f)(C),C (*df)(C) )` pour qui retourne la racines de l'équation $f(z) = 0$, limite des itérations de Newton : $z_{n+1} = z_n - \frac{f(z_n)}{df(z_n)}$ avec par exemple $f(z) = z^p - 1$ et $df(z) = f'(z) = p * z^{p-1}$, partant de $z0$. Cette fonction retournera le nombre complexe nul en cas de non convergence.

Q 3 Faire un programme, qui appelle la fonction Newton pour les points de la grille $z_0 + dn + dmi$ pour $(n, m) \in \{0, \ldots, N-1\} \times \{0, \ldots, M-1\}$ où les variables z_0, d, N, M sont données par l'utilisateur.

Afficher un tableau $N \times M$ résultat, qui nous dit vers quel racine la méthode de Newton a convergé en chaque point de la grille.

Q4 modifier les sources `http://www.ann.jussieu.fr/~hecht/ftp/cpp/mini-pj-fractale.tgz` afin d'affiché graphique, il suffit de modifier la fonction `void Draw()`, de plus les bornes de l'écran sont entières et sont stockées dans les variables global `int Width , Height;`.

Sous linux pour compiler et éditer de lien, il faut entrer les commandes `shell` suivantes :

```
g++ ex1.cpp -L/usr/X11R6/lib  -lGL -lGLUT -lX11 -o ex1
 # pour afficher le dessin dans une fenêtre X11.
```

ou mieux utiliser la commande unix `make` ou `gmake` en créant le fichier `Makefile` contenant :

```
CPP=g++
CPPFLAGS= -I/usr/X11R6/include
LIB= -L/usr/X11R6/lib -lglut -lGLU -lGL -lX11 -lm
%.o:%.cpp
(caractere de tabulation  -->/)$(CPP) -c $(CPPFLAGS) $^
cercle: cercle.o
(caractere de tabulation  -->/)g++ ex1.o  $(LIB) -o ex1
```

7.2 Les classes tableaux

Nous commencerons sur une version didactique, nous verrons la version complète qui est dans le fichier « tar compressé » `http://www.ann.jussieu.fr/~hecht/ftp/cpp/RNM-v3.tar.gz`.

7.2.1 Version simple d'une classe tableau

Mais avant toute chose, me paraît clair qu'un vecteur sera un classe qui contient au minimum la taille n, et un pointeur sur les valeurs. Que faut'il dans cette classe minimale (noté `A`).

```
typedef double K;                                          //    définition du corps
class A { public:                                            //    version 1 -------

  int n;                                                  //    la taille du vecteur
  K *v;                                                //    pointeur sur les n valeurs
  A() { cerr <<" Pas de constructeur pas défaut " << endl; exit(1);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                  //    constructeur
  ~A() {delete [] v;}                                        //    destructeur
   K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
};
```

Cette classe ne fonctionne pas car le constructeur par copie par défaut fait une copie bit à bit et donc le pointeur `v` est perdu, il faut donc écrire :

```
class A { public:                                            //    version 2 -------
  int n;                                                  //    la taille du vecteur
  K *v;                                                //    pointeur sur les n valeurs
  A() { cerr <<" Pas de constructeur pas défaut " << endl; exit(1);}
  A(const A& a) :n(a.n),v(new K[a.n])                 //    constructeur par copie
       { operator=(a);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                  //    constructeur
  A& operator=(A &a) {                                       //    copie
        assert(n==a.n);
        for(int i=0;i<n;i++) v[i]=a.v[i];
        return *this;}
  ~A() {delete [] v;}                                        //    destructeur
  K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
};
```

Maintenant nous voulons ajouter les opérations vectorielles $+, -, *, ...$

```
class A { public:                                            //    version 3 -------
  int n;                                                  //    la taille du vecteur
  K *v;                                                //    pointeur sur les n valeurs
  A() { cerr <<" Pas de constructeur pas défaut " << endl; exit(1);}
  A(const A& a) :n(a.n),v(new K[a.n]) { operator=(a);}
  A(int i) : n(i),v(new K[i]) { assert(v);}                  //    constructeur
  A& operator=(A &a) {assert(n==a.n);
                      for(int i=0;i<n;i++) v[i]=a.v[i];
                      return *this}
  ~A() {delete [] v;}                                        //    destructeur
  K & operator[](int i) const  {assert(i>=0 && i <n);return v[i];}
```

```
  A operator+(const &a) const;                                    //    addition
  A operator*(K &a) const;                          //    espace vectoriel à droite

 private:                    //    constructeur privé pour faire des optimisations
  A(int i, K* p) : n(i),v(p){assert(v);}
  friend A operator*(const K& a,const A& );       //    multiplication à gauche

 };
```

Il faut faire attention dans les paramètres d'entrée et de sortie des opérateurs. Il est clair que l'on ne veut pas travailler sur des copies, mais la sortie est forcément un objet et non une référence sur un objet car il faut allouer de la mémoire dans l'opérateur et si l'on retourne une référence aucun destructeur ne sera appelé et donc cette mémoire ne sera jamais libérée.

```
                  //   version avec avec une copie du tableau au niveau du return
A A::operator+(const A & a) const {
 A b(n); assert(n == a.n);
 for (int i=0;i<n;i++) b.v[i]= v[i]+a.v[i];
 return b;                          //    ici l'opérateur A(const A& a) est appeler
}
```

Pour des raisons optimisation nous ajoutons un nouveau constructeur `A(int,K*)` qui évitera de faire une copie du tableau.

```
                                    //    --- version optimisée sans copie----
A A::operator+(const A & a) const {
 K  *b(new K[n]); assert(n == a.n);
 for (int i=0;i<n;i++) b[i]= v[i]+a.[i];
 return A(n,b);                //    ici l'opérateur A(n,K*) ne fait pas de copie
}
```

Pour la multiplication par un scalaire à droite on a :

```
                                                //    --- version optimisée ----
A A::operator*(const K & a) const {
 K  *b(new K[n]); assert(n == a.n);
 for (int i=0;i<n;i++) b[i]= v[i]*a;
 return A(n,b);                //    ici l'opérateur A(n,K*) ne fait pas de copie
}
```

Pour la version à gauche, il faut définir `operator*` extérieurement à la classe car le terme de gauche n'est pas un vecteur.

```
A operator*(const K & a,const A & c) {
 K  *b(new K[c.n]);
 for (int i=0;i<c.n;i++) b[i]= c[i]*a;
 return A(n,b);                    //    attention cette opérateur est privé donc
                        //    cette fonction doit est ami ( friend) de la classe
}
```

Maintenant regardons ce qui est exécuté dans le cas d'une expression vectorielle.

```
int n=100000;
A a(n),b(n),c(n),d(n);
...                                          //    initialisation des tableaux a,b,c
d = (a+2.*b)+c*2.0;
```

voilà le pseudo code généré avec les 3 fonctions suivantes : `add(a,b,ab)`, `mulg(s,b,sb)`, `muld(a,s,as)`, `copy(a,b)` où le dernier argument retourne le résultat.

```
A a(n),b(n),c(n),d(n);
A t1(n),t2(n),t3(n),t4(n);
muld(2.,b),t1);                                                  //    t1 = 2.*b
add(a,t1,t2);                                                   //    t2 = a+2.*b
mulg(c,2.,t3);                                                   //    t3 = c*2.
add(t2,t3,t4);                                          //    t4 = (a+2.*b)+c*2.0;
copy(t4,d);                                              //    d = (a+2.*b)+c*2.0;
```

Nous voyons que quatre tableaux intermédiaires sont créés ce qui est excessif. D'où l'idée de ne pas utiliser toutes ces possibilités car le code généré sera trop lent.

Remarque 2. *Il est toujours possible de créer des classes intermédiaires pour des opérations prédéfinies afin obtenir se code générer :*

```
A a(n),b(n),c(n),d(n);
for (int i;i<n;i++)
  d[i] = (a[i]+2.*b[i])+c[i]*2.0;
```

Ce cas me paraît trop compliquer, mais nous pouvons optimiser raisonnablement toutes les combinaisons linéaires à 2 termes.

Mais, il est toujours possible d'éviter ces problèmes en utilisant les opérateurs `+=`, `-=`, `*=`, `/=` ce que donnerai de ce cas

```
A a(n),b(n),c(n),d(n);
for (int i;i<n;i++)
   d[i] = (a[i]+2.*b[i])+c[i]*2.0;
```

D'où l'idée de découper les classes vecteurs en deux types de classe les classes de terminer par un _ sont des classes sans allocation (cf. new), et les autres font appel à l'allocateur `new` et au déallocateur `delete`. De plus comme en fortran 90, il est souvent utile de voir une matrice comme un vecteur ou d'extraire une ligne ou une colonne ou même une sous-matrice. Pour pouvoir faire tous cela comme en fortran 90, nous allons considèrer un tableau comme un nombre d'élément n, un incrément s et un pointeur v et du tableau suivant de même type afin de extraire des sous-tableau.

7.2.2 les classes RNM

Nous définissons des classes tableaux à un, deux ou trois indices avec ou sans allocation. Ces tableaux est défini par un pointeur sur les valeurs et par la forme de l'indice (class `ShapeOfArray`) qui donne la taille, le pas entre de valeur et le tableau suivant de même type (ces deux données supplémentaire permettent extraire des colonnes ou des lignes de matrice, ...).

La version est dans le fichier « tar compressé » `http://www.ann.jussieu.fr/~hecht/ftp/cpp/RNM-v3.tar.gz`.
Nous voulons faire les opérations vectorielles classiques sur `A, C,D` tableaux de type `KN<R>` suivante par exemples :

```
A = B; A += B; A -= B; A = 1.0; A = 2*.C;
A = A+B; A = 2.0*C+ B; C = 3.*A-5*B;
 A /= C;  A *=C;                                          //    .. non standard ..
R c = A[i];
A[j] = c;
```

Pour des raisons évidentes nous ne sommes pas allés plus loin que des combinaisons linéaires à plus de deux termes. Toutes ces opérations sont faites sans aucune allocation et avec une seule boucle.
De plus nous avons défini, les tableaux 1,2 ou 3 indices, il est possible extraire une partie d'un tableau, une ligne ou une colonne.
Attention, il y a deux types de classe les classes avec un _ en fin de nom, et les autre sans. Les classes avec un _ en fin de nom, ne font aucune opération d'allocation ou destruction de mémoire dynamique, c'est-à-dire qui n'y a aucun opérateur `new` ou `delete` dans le code associé. Ces classes tableaux peuvent être vue comme une généralisation de la notion de références sur un tableau. Elle permet de donner un nom a une tranche d'un tableau, comme par exemple :

```
KN<double> a(10);                  //    un tableau de 10 valeurs qui est alloué
a=100.;                                 //    met tous les 10 valeurs de a à 100.
KN_<double> b(a);          //    le tableau de 10 valeurs avec le même pointeur
KN<double>  c(a);                  //    un autre tableau qui est la copie de a.
                                                                  //    la preuve
a[1]=1.;                                    //    => b[1] == 1. car &a[i] == &b[i]
                                 //    mais bien sur c[1] == 100. car &c[i] != &a[i]
```

Voilà le début du code de la classe `KN_` :

```
template<class R>
class KN_: public  ShapeOfArray {
protected:
  R *v;                                  //    le pointeur du tableau v[i × s], i ∈ [0..N()[
public:
  typedef R K;                                                    //    type of data
                                       //    les constructeurs a partir d'un pointeur
  KN_(R *u,long nn);                                              //    pointeur + n
  KN_(R *u,long nn,long s);          //    pointeur + n + valeurs avec un pas de s
  KN_(const KN_<R> & u);                          //    le constructeur par copy

                                                                  //    des méthodes
  long N() const {return n;}                              //    indice dans ı ∈ [0..N()[
  bool unset() const { return !v;}
  void set(R * vv,int nn,int st=1,int nx=-1) {v=vv;n=nn;step=st;next=nx;}
  long size() const{return step?n*step:n;}                        //    taille alloué

  //    remarque s est le pas (step) du tableau (généralement s = 1)
  R min() const;                                  //    min_{i∈[0..N()[} v[i s]
```

```
  R max() const;                                  //      max_{i∈[0..N()[} v[i s]
  R sum() const;                                  //      Σ_{i∈[0..N()[} v[i s]
  double norm() const;                            //      Σ_{i∈[0..N()[} |v[i s]|^2
  double l2() const;                              //      ( Σ_{i∈[0..N()[} v[i s]^2 )^{1/2}
  double l1() const;                              //      Σ_{i∈[0..N()[} |v[i s]|
  double linfty() const;                          //      max_{i∈[0..N()[} |v[i s]|
  double lp(double p) const;                      //      ( Σ_{i∈[0..N()[} v[i s]^p )^{1/p}

  //    pour conversion en pointeur
  operator R *() const {return v;}                //    transforme un KN_ en pointeur

 .
  KN_ operator()(const SubArray & sa) const     //    la fonction pour prendre un
       { return KN_(*this,sa);}     //   sous tableau (SubArray(N,debut=0,pas=1))

...  + tous les opérateurs  =, +=, -=, *=,  /=

 //    Operation avec des matrices VirtualMatrice
 KN_& operator =(const typename VirtualMatrice<R>::plusAx & Ax)
    {*this=R();             //    mise à 0 (le constructeur R() rend 0 de type R)
     Ax.A->addMatMul(Ax.x,*this);return *this;}

 KN_& operator +=(const typename VirtualMatrice<R>::plusAx & Ax)
    {  Ax.A->addMatMul(Ax.x,*this);return *this;}

.... etc
};
```

Maintenant, Voilà les constructeurs et le destructeur, les fonctions d'allocations de la classe KN qui dérive de KN_ :

```
template<class R>
class KN :public KN_<R> { public:
  typedef R K;
  KN() : KN_<R>(0,0) {}   //    constructeur par défaut tableau le longueur nulle
  KN(long nn);                                  //    construit un KN de longueur nn
  KN(long nn, R * p); //    construit un KN de longueur nn initiales le tableau p
  KN(long nn,R (*f)(long i) );                      //    construit KN (f(i))_{i=[0..nn[}
  KN(long nn,const  R & a);                        //    construit un KN (a)_{i=[0..nn[}
  template<class S>   KN(const KN_<S> & s);            //    construit par copie
  template<class S>
       KN(const KN_<S> & s,R (*f)(S ));             //    construit (f(s[i]))_{i=[0..s.N()[}
  KN(const KN<R> & u);                                 //    construit par copie
   void resize(long nn);                       //    pour redimensionner le tableau
  ~KN(){delete [] this->v;}                                  //    le destructeur
  ...
  };
```

Et voilà de la classe `VirtualMatrice` qui modélise, une matrice. Une matrice sera une classe dérive de `VirtualMatrice`, qui aura définit la méthode `void addMatMul( const KN_<R> & x, K` qui ajoute à `y` le produit matrice vecteur.

```
template<class R>
struct  VirtualMatrice { public:
  virtual void addMatMul(const KN_<R> &  x, KN_<R> & y) const =0;
   ...
                              //    pour stocker les données de l'opération += A*x
   struct  plusAx { const VirtualMatrice * A; const KN_<R> &  x;
   plusAx( const VirtualMatrice * B,const KN_<R> &  y) :A(B),x(y) {} };

  virtual ~VirtualMatrice(){}
};
```

Voilà, un example très complet des possibilité de classes.

```
#include<RNM.hpp>

 ....

  typedef double R;
  KNM<R> A(10,20);                                                    //    un matrice
  . . .
  KN_<R> L1(A(1,'.');                                 //    la ligne 1 de la matrice A;
  KN<R> cL1(A(1,'.');                       //    copie de la ligne 1 de la matrice A;
  KN_<R> C2(A('.',2);                                 //    la colonne 2 de la matrice A;
  KN<R> cC2(A('.',2);                     //    copie de la colonne 2 de la matrice A;
  KNM_<R> pA(FromTo(2,5),FromTo(3,7));      //    partie de la matrice A(2:5,3:7)
                                                      //    vue comme un matrice 4x5

  KNM B(n,n);
  B(SubArray(n,0,n+1))                                //    le vecteur diagonal de B;
  KNM_ Bt(B.t());                                   //    la matrice transpose sans copie
```

Pour l'utilisation, utiliser l'ordre `#include "RNM.hpp"`, et les flags de compilation `-DCHECK_KN` ou en définisant la variable du preprocesseur `cpp` du `C++` avec l'ordre `#defined CHECK_KN`, avant la ligne include.
Les définitions des classes sont faites dans 4 fichiers `RNM.hpp`, `RNM_tpl.hpp`, `RNM_op.hpp`, `RNM_op.hpp`.
Pour plus de détails voici un exemple d'utilisation assez complet.

7.2.3 Exemple d'utilisation

```
namespace std

#define CHECK_KN
#include "RNM.hpp"
#include "assert.h"
```

```
using namespace std;
          //    definition des 6 types de base des tableaux a 1,2 et 3 parametres
typedef double R;
typedef KN<R> Rn;
typedef KN_<R> Rn_;
typedef KNM<R> Rnm;
typedef KNM_<R> Rnm_;
typedef KNMK<R> Rnmk;
typedef KNMK_<R> Rnmk_;
R f(int i){return i;}
R g(int i){return -i;}
int main()
{
  const int n= 8;
  cout << "Hello World, this is RNM use!" << endl << endl;
  Rn a(n,f),b(n),c(n);
  b =a;
  c=5;
  b *= c;

  cout << " a = " << (KN_<const_R>) a << endl;
  cout << " b = " <<  b << endl;

                                                    //    les operations vectorielles
  c = a + b;
  c = 5. *b + a;
  c = a + 5. *b;
  c = a - 5. *b;
  c = 10.*a - 5. *b;
  c = 10.*a + 5. *b;
  c += a + b;
  c += 5. *b + a;
  c += a + 5. *b;
  c += a - 5. *b;
  c += 10.*a - 5. *b;
  c += 10.*a + 5. *b;
  c -= a + b;
  c -= 5. *b + a;
  c -= a + 5. *b;
  c -= a - 5. *b;
  c -= 10.*a - 5. *b;
  c -= 10.*a + 5. *b;


  cout <<" c = " << c << endl;
  Rn u(20,f),v(20,g);                                    //    2 tableaux u,v de 20
                                                  //   initialiser avec u_i = f(i), v_j = g(i)
  Rnm A(n+2,n);                                          //    Une matrice n+2 x n

  for (int i=0;i<A.N();i++)                                                  //    ligne
     for (int j=0;j<A.M();j++)                                             //    colonne
       A(i,j) = 10*i+j;

  cout << "A=" << A << endl;
  cout << "Ai3=A('.', 3 ) = " << A('.', 3 ) << endl;          //    la colonne 3
  cout << "A1j=A( 1 ,'.') = " << A( 1 ,'.') << endl;            //    la ligne 1
  Rn CopyAi3(A('.', 3 ));                              //    une copie de la colonne 3
```

```
cout << "CopyAi3 = " << CopyAi3;

Rn_ Ai3(A('.', 3 ));                          //    la colonne 3 de la matrice
CopyAi3[3]=100;
cout << CopyAi3 << endl;
cout << Ai3 << endl;

assert( & A(0,3) == & Ai3(0));                //    verification des adresses


Rnm S(A(SubArray(3),SubArray(3)));                      //    sous matrice 3x3

Rn_ Sii(S,SubArray(3,0,3+1));         //    la diagonal de la matrice sans copy

cout << "S=  A(SubArray(3),SubArray(3)  = " << S <<endl;
cout << "Sii     = " << Sii <<endl;
b = 1;
                                                //    Rn Ab(n+2) = A*b; error
Rn Ab(n+2);
Ab = A*b;
cout << " Ab = A*b  =" << Ab << endl;

Rn_ u10(u,SubArray(10,5));                //    la partie [5,5+10[ du tableau u
cout << "u10 " << u10 << endl;
v(SubArray(10,5)) += u10;
cout << " v = " << v << endl;
cout << " u(SubArray(10))    "  << u(SubArray(10)) << endl;
cout << " u(SubArray(10,5))  "  << u(SubArray(10,5)) << endl;
cout << " u(SubArray(8,5,2))"  << u(SubArray(8,5,2))
     << endl;


cout << " A(5,'.')[1] " << A(5,'.')[1] << " " << " A(5,1) = "
     << A(5,1) << endl;
cout << " A('.',5)(1) = "<< A('.',5)(1) <<  endl;
cout << " A(SubArray(3,2),SubArray(2,4)) = " << endl;
cout <<  A(SubArray(3,2),SubArray(2,4)) << endl;
A(SubArray(3,2),SubArray(2,4)) = -1;
A(SubArray(3,2),SubArray(2,0)) = -2;
cout << A << endl;

Rnmk B(3,4,5);
for (int i=0;i<B.N();i++)                                    //    ligne
   for (int j=0;j<B.M();j++)                                //    colonne
       for (int k=0;k<B.K();k++)                              //    ....
          B(i,j,k) = 100*i+10*j+k;
cout << " B      = " << B << endl;
cout << " B(1  ,2  ,'.') " << B(1  ,2  ,'.') << endl;
cout << " B(1  ,'.',3  ) " << B(1  ,'.',3  ) << endl;
cout << " B('.',2  ,3  ) " << B('.',2  ,3  ) << endl;
cout << " B(1  ,'.','.') " << B(1,'.'  ,'.') << endl;
cout << " B('.',2  ,'.') " << B('.',2  ,'.') << endl;
cout << " B('.','.',3  ) " << B('.','.',3  ) << endl;

cout << " B(1:2,1:3,0:3)   = "
     << B(SubArray(2,1),SubArray(3,1),SubArray(4,0)) << endl;
```

```
                                                //    copie du sous tableaux
  Rnmk Bsub(B(FromTo(1,2),FromTo(1,3),FromTo(0,3)));
  B(SubArray(2,1),SubArray(3,1),SubArray(4,0))  = -1;
  B(SubArray(2,1),SubArray(3,1),SubArray(4,0)) += -1;
  cout << " B      = " << B << endl;
  cout << Bsub << endl;

  return 0;
}
```

7.2.4 Un resolution de systéme linéaire avec le gradient conjugué

L'algorithme du gradient conjugué présenté dans cette section est utilisé pour résoudre le système
linéaire $Ax = b$, où A est une matrice symétrique positive $n \times n$.
Cet algorithme est basé sur la minimisation de la fonctionnelle quadratique $E : \mathbb{R}^n \to \mathbb{R}$ suivante :

$$E(x) = \frac{1}{2}(Ax, x) - (b, x),$$

avec un changement de variable $y = Cx$ où $(.,.)_C$ est le produit scalaire associer à une une matrice C, symétrique définie positive de $\mathbb{R}^n$.
C'est aussi la minimisation de la fonctionnelle :

$$E_c(y) = \frac{1}{2}(ACy, y)_C - (b, y)_C,$$

car la matrice CAC est symétrique positive.

Algorithme 6. **Le gradient conjugué preconditionné**

soient $x^0 \in \mathbb{R}^n$, ε, C donnés
$G^0 = Ax^0 - b$
$H^0 = -CG^0$
— *pour* $i = 0$ *à* n
$\rho = -\dfrac{(G^i, H^i)}{(H^i, AH^i)}$
$x^{i+1} = x^i + \rho H^i$
$G^{i+1} = G^i + \rho AH^i$
$\gamma = \dfrac{(G^{i+1}, G^{i+1})_C}{(G^i, G^i)_C}$
$H^{i+1} = -CG^{i+1} + \gamma H^i$
si $(G^{i+1}, G^{i+1})_C < \varepsilon$ *stop*

Théorème 7.1. *Notons* $\mathcal{E}_C(x) = \sqrt{(Ax-\overline{x}, x-\overline{x})_C}$, *l'erreur dans la norme de* A *preconditionné, où* $\overline{x}$ *est la solution du problème. Alors l'erreur à l'intération* k *un gradient conjugué est majoré :*

$$\mathcal{E}_C(x^k) \leq 2\left(\frac{\sqrt{K_C(A)-1}}{\sqrt{K_C(A)+1}}\right)^k \mathcal{E}_C(x^0)$$

où $K_C(A)$ *est le conditionnement de la matrice* CA, *c'est à dire* $K_C(A) = \frac{\lambda_1^C}{\lambda_n^C}$ *où* λ_1^C *(resp.* λ_n^C*) est la plus petite (resp. grande) valeur propre de matrice* CA .

Le démonstration est technique et est faite dans [**?**, chap 8.3]. Voila comment écrire un gradient conjugué avec ces classes.

7.2.5 Gradient conjugué préconditionné

Listing 5: *(GC.hpp)*

```
                //    exemple de programmation du gradient conjugue precondiconnée
template<class R,class M,class P>
int GradienConjugue(const M & A,const P & C,const KN_<R> &b,KN_<R> &x,
                    int nbitermax,double eps)
{
   int n=b.N();
   assert(n==x.N());
   KN<R> g(n), h(n), Ah(n), & Cg(Ah);          //    on utilise Ah pour stocke Cg
   g = A*x;
   g -= b;                                                    //    g = Ax-b
   Cg = C*g;                                          //    gradient preconditionne
   h =-Cg;
   R g2 = (Cg,g);
   R reps2 = eps*eps*g2;                                    //    epsilon relatif
   for (int iter=0;iter<=nbitermax;iter++)
     {
       Ah = A*h;
       R ro =  - (g,h)/ (h,Ah);          //    ro optimal (produit scalaire usuel)
       x += ro *h;
       g += ro *Ah;      //    plus besoin de Ah, on utilise avec Cg optimisation
       Cg = C*g;
       R g2p=g2;
       g2 = (Cg,g);
       cout << iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
       if (g2 < reps2) {
          cout << iter <<  "  ro = " << ro << " ||g||^2 = " << g2 << endl;
          return 1;                                                    //    ok
          }
       R gamma = g2/g2p;
       h *= gamma;
       h -= Cg;                                          //    h = -Cg * gamma* h
     }
   cout << " Non convergence de la méthode du gradient conjugue " <<endl;
   return 0;
}
//    la matrice Identite --------------------
template <class R>
```

```
class MatriceIdentite: VirtualMatrice<R> { public:
 typedef  VirtualMatrice<R>::plusAx plusAx;
 MatriceIdentite(int n):VirtualMatrice<R> (n) {};
 void addMatMul(const  KN_<R>  & x, KN_<R> & Ax) const {   Ax+=x; }
 plusAx operator*(const KN<R> &  x) const {return plusAx(this,x);}
};
```

7.2.6 Test du gradient conjugué

Pour finir voilà, un petit programme pour tester le GC sur cas différent. Le troisième cas étant la résolution de l'équation différencielle : $-u'' = 1$ sur $[0, 1]$ avec comme conditions aux limites $u(0) = u(1) = 0$, par la méthode de l'élément fini. La solution exact est $f(x) = x(1 - x)/2$, nous verifions donc l'erreur sur le resultat.

Listing 6: *(GradConjugue.cpp)*

```
#include <fstream>
#include <cassert>
#include <algorithm>

using namespace std;

#define KN_CHECK
#include "RNM.hpp"
#include "GC.hpp"

typedef double R;
class MatriceLaplacien1D: VirtualMatrice<R> { public:
 MatriceLaplacien1D(int n) :VirtualMatrice<R>(n) {};
 void addMatMul(const KN_<R> & x, KN_<R> & Ax) const;
 plusAx operator*(const KN<R> &  x) const {return plusAx(*this,x);}
};

 void MatriceLaplacien1D::addMatMul(const KN_<R> & x, KN_<R> & Ax) const {
  int n= x.N(),n_1=n-1;
  double h=1./(n_1), h2= h*h, d = 2/h, d1 = -1/h;
  R Ax0=Ax[0], Axn_1=Ax[n_1];
  Ax=0;
  for (int i=1;i< n_1; i++)
    Ax[i] = (x[i-1] +x[i+1]) * d1 + x[i]*d ;

                                                                    //     CL
   Ax[0]=x[0];
   Ax[n_1]=x[n_1];
  }


int main(int argc,char ** argv)
{
 typedef KN<double> Rn;
 typedef KN_<double> Rn_;
 typedef KNM<double> Rnm;
 typedef KNM_<double> Rnm_;
```

```
{
 int n=10;
 Rnm A(n,n),C(n,n),Id(n,n);
 A=-1;
 C=0;
 Id=0;
 Rn_ Aii(A,SubArray(n,0,n+1));        //   la diagonal de la matrice A sans copy
 Rn_ Cii(C,SubArray(n,0,n+1));        //   la diagonal de la matrice C sans copy
 Rn_ Idii(Id,SubArray(n,0,n+1));     //   la diagonal de la matrice Id sans copy
 for (int i=0;i<n;i++)
    Cii[i]= 1/(Aii[i]=n+i*i*i);
 Idii=1;
 cout << A;
 Rn x(n),b(n),s(n);
 for (int i=0;i<n;i++) b[i]=i;
 cout << "GradienConjugue preconditionne par la diagonale " <<  endl;
 x=0;
 GradienConjugue(A,C,b,x,n,1e-10);
 s = A*x;
 cout << " solution : A*x= "  << s << endl;
 cout << "GradienConjugue preconditionnee par la identity  " << endl;
 x=0;
 GradienConjugue(A,MatriceIdentite<R>(n),b,x,n,1e-6);
 s = A*x;
 cout << s << endl;
}
{
 cout << "GradienConjugue laplacien 1D  par la identity  " << endl;
 int N=100;
 Rn b(N),x(N);
 R h= 1./(N-1);
 b= h;
 b[0]=0;
 b[N-1]=0;
 x=0;
 R t0=CPUtime();
 GradienConjugue(MatriceLaplacien1D(n),MatriceIdentite<R>(n) ,b,x,N,1e-5);
 cout << " Temps cpu = " << CPUtime() - t0<< "s" << endl;
 R err=0;
 for (int i=0;i<N;i++)
   {
   R xx=i*h;
   err= max(fabs(x[i]- (xx*(1-xx)/2)),err);
   }
 cout << "Fin err=" << err << endl;
}
 return 0;
}
```

7.2.7 Sortie du test

```
10x10   :
          10  -1  -1  -1  -1  -1  -1  -1  -1  -1
          -1  11  -1  -1  -1  -1  -1  -1  -1  -1
          -1  -1  18  -1  -1  -1  -1  -1  -1  -1
          -1  -1  -1  37  -1  -1  -1  -1  -1  -1
          -1  -1  -1  -1  74  -1  -1  -1  -1  -1
          -1  -1  -1  -1  -1 135  -1  -1  -1  -1
          -1  -1  -1  -1  -1  -1 226  -1  -1  -1
          -1  -1  -1  -1  -1  -1  -1 353  -1  -1
          -1  -1  -1  -1  -1  -1  -1  -1 522  -1
          -1  -1  -1  -1  -1  -1  -1  -1  -1 739
        GradienConjugue preconditionne par la diagonale
6  ro = 0.990712 ||g||^2 = 1.4253e-24
 solution : A*x= 10     :    1.60635e-15  1 2 3 4 5 6 7 8 9

GradienConjugue preconditionnee par la identity
9  ro = 0.0889083 ||g||^2 = 2.28121e-15
10      :       6.50655e-11    1 2 3 4 5 6 7 8 9

GradienConjugue laplacien 1D  preconditionnee par la identity
48  ro = 0.00505051 ||g||^2 = 1.55006e-32
 Temps cpu = 0.02s
Fin err=5.55112e-17
```

Exercice 12.

Modifier, l'exemple `GradConjugue.cpp`, pour résoudre le problème suivant, trouver $u(x,t)$ une solution

$$\frac{\partial u}{\partial t} - \Delta u = f; \quad \text{dans }]0,L[$$

$$\text{pour } t=0, u(x,0) = u_0(x) \quad \text{et } u(0,t) = u(L,t) = 0$$

en utilisant un θ schéma pour discrétiser en temps, c'est à dire que

$$\frac{u^{n+1}-u^n}{\Delta t} - \Delta\big(\theta u^{n+1} + (1-\theta)u^n\big) = f; \quad \text{dans }]0,L[$$

où u^n est une approximation de $u(x, n\Delta t)$, faite avec des éléménts finis P_1, avec un maillage régulier de $]0,L|$ en M éléments. les fonctions élémentaires seront noté w^i, avec pour $i=0,...,M$, avec $x_i = \frac{i/N}{L}$

$$w^i|_{]x_{i-1},x_i[\cup]0,L[} = \frac{x-x_i}{x_{i-1}-x_i}, \quad w^i|_{]x_i,x_{i+1}[\cup]0,L[} = \frac{x-x_i}{x_{i+1}-x_i}, \quad w^i|_{]0,L[\setminus]x_{i-1},x_{i+1}[} = 0$$

C'est a dire qu'il faut commencer par construire du classe qui modélise la matrice

$$\mathcal{M}_{\alpha\beta} = \left(\int_{]O,L[} \alpha w^i w^j + \beta (w^i)'(w^j)' \right)_{i=1...M, j=1...M}$$

en copiant la classe `MatriceLaplacien1D`.
Puis, il suffit, d'approcher le vecteur $F = (f_i)_{i=0..M} = \left(\int_{]O,L[} f w^i \right)_{i=0..M}$ par le produit $F_h = \mathcal{M}_{1,0} \left(f(x_i) \right)_{i=1,M}$, ce qui revient à remplacer f par

$$f_h = \sum_i f(x_i) w^i$$

.

Q1 Ecrire l'algorithme mathematiquement, avec des matrices

Q2 Trancrire l'algorithme

Q3 Visualiser les résultat avec gnuplot, pour cela il suffit de crée un fichier par pas de temps contenant la solution, stocker dans un fichier, en inspirant de

```
#include<sstream>
#include<ofstream>
#include<iostream>
  ....
  stringstream ff;
  ff << "sol-"<< temps << ends;
  cout << " ouverture du fichier" <<ff.str.c_str()<< endl;
  {
     ofstream file(ff.str.c_str());
     for (int i=0;i<=M;++i)
       file <<x[i]<< endl;
   }          //    fin de bloc => destruction de la variable file
                                   //    => fermeture du fichier
  ...
```

7.3 Des classes pour les Graphes

Les sources sont disponibles : `http://www.ann.jussieu.fr/~hecht/ftp/cpp/Graphe.tar.bz2`.

7.4 Définition et Mathématiques

- On définit un graphe $\mathbf{G}$ comme étant un couple (X, G) où X est un ensemble appelé l'ensemble des sommets et G un sous ensemble de X^2 appelé l'ensemble des arcs ou arêtes orientées.
- une arête est un arc qui a perdu son orientation
- une chaine est une suite d'arêtes $(\mu_i, (i = 1, n))$ de G telle que les arêtes μ_i et μ_{i-1} aient un sommet s_i en commun si $i > 0$ et $i < n$ et telle que pour $i > 0$ et $i < n$, l'arête μ_i ait pour autre sommet le sommet s_{i+1}. Soit s_o (resp. s_n) le sommet de μ_1 (resp. μ_n) non dans μ_2 (resp. μ_{n-1}). Si la chaine est réduite a une arête μ_1, s_o et s_1 seront les deux sommets de l'arête μ_1.
- On définit la relation d'équivalence sur les sommets de $\mathbf{G}$ par : $x \equiv y$ il existe une chaine reliant x à y.
- les composantes connexes de $\mathbf{G}$ seront les classes d'équivalence de la relation $\equiv$.
- un cycle est une chaine fermée (le sommet s_o est égale à s_n) telle que tous les arcs soient distincts.
- à chaque cycle on peut associer un vecteur μ de $\mathbb{R}^G$ dont les composantes sont définies par

$$\mu_i = \begin{cases} +0 & \text{si l'arc } i \text{ n'est pas dans le cycle,} \\ +1 & \text{l'arc } i \text{ a le même sens que le sens du parcours du cycle,} \\ -1 & \text{sinon.} \end{cases} \tag{3}$$

  par abus de langage, on confondra souvent un cycle et son vecteur associé. On notera par $V(\mathbf{G})$ l'espace vectoriel engendré par les cycles de $\mathbf{G}$.

Théorème 7.1. *Soit* $\mathbf{G}$ *un graphe à* n *sommets,* m *arcs et* p *composantes connexes, alors la dimension de* $V(\mathbf{G})$ *est égal à* $m - n + p$.

Démonstration . voir Berge [3]

Théorème 7.2. *Définition Soit* $\mathbf{H} = (X, H)$ *un graphe. Les propriétés suivantes sont équivalentes et caractérisent un arbre (*n *nombre de sommets,* m *le nombre d'arcs).*

1. $\mathbf{H}$ *est connexe, sans cycle*
2. $\mathbf{H}$ *est connexe et* $m = n - 1$
3. $\mathbf{H}$ *est sans cycle et* $m = n - 1$
4. $\mathbf{H}$ *est sans cycle et en ajoutant un arc quelconque on crée un cycle unique*
5. $\mathbf{H}$ *est connexe et si l'on supprime un arc quelconque il n'est plus connexe.*

Démonstration. voir Berge [3]

Théorème 7.3. *Définition Soit* $\mathbf{G} = (X, G)$ *un graphe connexe alors il existe un sous graphe* $\mathbf{H} = (X, H)$ *tel que* H *soit inclus dans* G *et tel que* $\mathbf{H}$ *soit un arbre.* $\mathbf{H}$ *sera appelé un arbre maximal de* $\mathbf{G}$.

Démonstration. Soit $\mathbf{H} = (X, H)$ un graphe connexe tel que pour tout sous ensemble H' de H et différent de H, le graphe (X, H') est non connexe (il en existe car le graphe (X, G) est connexe, et les ensembles G et X sont finis). II est alors clair que $\mathbf{H}$ est un arbre d'après la propriété v) du théorème A.2. ∎

Théorème 7.4. *Soit* $\mathbf{G} = (X, G)$ *un graphe connexe et soit* $\mathbf{H} = (X, H)$ *un arbre maximal de* G*, si* i *est un arc de* $\mathbf{G}$ *et non de* $\mathbf{H}$ *son adjonction à* H *détermine un cycle* γ^i *unique tel que son vecteur associé* $(\gamma^i_j)_{j\in G}$ *vérifie :* γ^i_i *est égal à 1. L'ensemble* $\{\gamma^i / i \in G\backslash H\}$ *forme une base de* $V(\mathbf{G})$*.*

Démonstration. Berge [3]
Soit $\mathbf{G} = (X, G)$ un graphe, on appelera un flot un vecteur $(\alpha_i)_{i\in G}$ tel que pour tout sommet s de X on ait

$$\sum_{i\in I_s^+} \alpha_i - \sum_{i\in I_s^-} \alpha_i = 0; \tag{4}$$

où I_s^+ (resp. I_s^-) est l'ensemble des arcs (x, y) de G tel que s soit égal à x (resp. y) ce qui est identique à l'ensemble des arcs partant (resp. arrivant) de s.

Remarque 3. *L'équation (4) nous dit que pour un flot tout ce qui entre en un sommet en ressort.*

Proposition A.1. l'ensemble des flots d'un graphe sans cycle est réduit à $\{0\}$.
Démonstration. Faisons une démonstration par récurrence sur l'ordre de $\mathbf{G}$ (nombre de sommets de $\mathbf{G}$). Si l'ordre de $\mathbf{G}$ est égal à un, on a trivialement la propriété. On suppose la propriété vraie pour tout graphe sans cycle d'ordre n.
Soit $\mathbf{G} = (X, G)$ un graphe sans cycle d'ordre $n + 1$ avec m arc. Soit $\alpha = (\alpha_i)_{i\in G}$ un flot de $\mathbf{G}$. Comme $\mathbf{G}$ est sans cycle, le théorème A.1 nous dit que $m - n + 1 < 0$, ce qui prouve qu'il existe un sommet j de G contenu dans une seule arête k de G. D'où d'après (4) , la composante α_k est nulle. On en déduit que $(\alpha_i)_{i\in G\backslash\{k\}}$ est un flot de $(X\backslash\{j\}, G\backslash\{k\})$. Il suffit d'appliquer l'hypothèse de récurrence pour montrer la propriété. ∎

Théorème 7.5. *l'espace vectoriel des flots est égal à l'espace vectoriel* $V(\mathbf{G})$ *engendré par les cycles du graphe* $\mathbf{G}$

Démonstration. On peut supposer $\mathbf{G}$ connexe car sinon on travaillera sur les composantes connexes de $\mathbf{G}$. Si $\mathbf{G}$ est connexe, il existe un arbre maximal $\mathbf{H} = (X, H)$ de $\mathbf{G} = (X, G)$. Soient les cycles (γ^i) associés à l'arbre pour $i \in G\backslash H$ (cf. théorème A.4). Il est clair que les vecteurs $(\gamma^i_j)_{\ j\in G}$ de $\mathbb{R}^G$ sont des flots. Donc tout vecteur de $V(\mathbf{G}$) est un flot. Soit $(\alpha_i)_{i\in G}$ un flot, on construit le flot $(\beta_j)_{j\in G}$ suivant

$$\beta_j = \alpha_j - \sum_{i\in G\backslash H} \alpha_j \gamma^i_j. \tag{5}$$

Comme par construction $\beta_j = 0$ pour tout $j \in G\backslash H$,on en déduit que le vecteur $(\beta_j)_{j\in H}$ est un flot de H. La proposition précédente et le théorème A.2 i) nous montrent que β_j est nul pour tout j dans H. D'où tout flot est somme de cycles ce qui prouve l'autre inclusion. ∎
Soit $\mathbf{G} = (X, G)$ un graphe connexe composé de n sommets et de m arêtes.

A chaque arc j on associe le vecteur $\mathbf{e}^j = (e_i^j)_{i\in X}$ de $\mathbb{R}^X$ tel que

$$e_i^j = \begin{cases} 0 & \text{si } i \text{ n'est pas un sommet de l'arc } j\,; \\ +1 & \text{si l'arc } j \text{ est de la forme } (i,k)\ (k\in X)\,; \\ -1 & \text{si l'arc } j \text{ est de la forme } (k,i)\ (k\in X). \end{cases} \tag{6}$$

7.5 Premiere Implémentation

Ici nous voulons, pourvoir calculer l'ordre des sommets d'un graphe, respectivement $o^{\pm}(s) = \#I^{\pm}(s)$ où les deux $I^{\pm}$ sont définies pour (4).
Puis, pour calculer les composantes connexes du graphe, nous aurons besoin de construire Les ensemble $I^+(s)\cup I^-(s)$. Pour, cela nous stockerons cette liste d' arcs dans la classe `Sommet`.

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


class Arc;
class Sommet;

class Sommet { public :
  int numero;                                          //    le numero du sommet
  int couleur;                                         //    la couleur du sommet
  int nbarcs;                                    //    nb d'arc contenant ce sommet

  Arc ** a;                                 //    listes des arcs contenant ce sommet
                                                       //    constructeur par default
  Sommet(int n=0) : numero(n),couleur(0),nbarcs(0),a(0) {}
  void Set(int i) {numero=i;}
  Sommet & operator=(int i) { numero=i;return *this;}
  int o() const {return nbarcs;}

                          //    pas d'operateurs de copies, c'est une class simple
                          //    dans le cas 2 on interdit la copie de sommet
  ~Sommet(){delete a;}
private:
  Sommet(const  Sommet &);
  void operator =(const  Sommet &);
};



class Arc {public:
  int numero;                                          //    le numero de l'arc
  Sommet *s[2];                    //    un tableau de 2 pointeurs sur les sommets
                                                       //    ( ici il faut refechir )

  Arc(): numero(0) {s[0]=s[1]=0; }               //    un constructeur par defaut
                                                       //    pour faire de tableau arc

                                                       //    les methodes ...
  const Sommet & operator[](int i) const
```

```
  { assert(i >=0 && i <=1); return (*s[i]);}

  Sommet & operator[](int i)
  { assert(i >=0 && i <=1); return (*s[i]);}


                                                    //    le vrai constructeur.
  void Set(int i,Sommet & a, Sommet & b) {numero=i;s[0]=&a; s[1]=&b;}

  const Sommet & SommetOppose(Sommet & si) const
  { assert(&si == s[0] || &si == s[1]);
  return  *(&si == s[0]? s[1] : s[0]); }

  Sommet & SommetOppose(Sommet & si)
  { assert(&si == s[0] || &si == s[1]);
  return  *(&si == s[0]? s[1] : s[0]); }


  Sommet * SommetOppose(Sommet * si) const
  { assert(si == s[0] || si == s[1]);
  return  (si == s[0]? s[1] : s[0]); }

private:
                              //    pas de copie on a toujours l'arc d'un graphee
  Arc(const  Arc & );
  void operator =(const  Arc & );
};




class Graphe { public:
  int n;                                          //    n nombre de sommets
  int m;                                              //    m nombre d'arcs
  Sommet * s;                                   //    le tableau des sommets
  Arc * a;                                        //    le tableau des Arcs


  Arc & operator[](int i){ return a[CheckA(i)];}
  const Arc & operator[](int i) const { return a[CheckA(i)];}

  Sommet & operator()(int i){ return s[CheckS(i)];}
  const Sommet & operator()(int i) const { return s[CheckS(i)];}
                                  //    G[i] est l'arc i et G(i) est le sommet i
  int CheckA(int i) const { assert(i>=0 && i < m); return i;}
  int CheckS(int i) const { assert(i>=0 && i < n); return i;}

  void MiseAJour();
  Graphe(int  );                                                  //    un graphe c
  Graphe(const char * filename);                //    un graphe lue dans filename
  ~Graphe()
  {
    cout << " delete Graphe: nb sommets " << n << " , nb arcs  " << m << endl;
    delete [] s;
    delete [] a;
```

```
  }

                                                    //    pas operateur de copie
private:
  Graphe(const Graphe &);
  Graphe & operator=(const Graphe &);
};



inline  ostream & operator<<(ostream & f, const Sommet &  s)
{
  if(&s) f << " s " << s.numero << " c " << s.couleur << " ";
  else cout << " s NULL ";
  return f;
}

inline ostream & operator<<(ostream & f, const Arc &  a)
{ f << " a " << a.numero << " :   " << a[0].numero << " "
    << a[1].numero << " "; return f;}


                        //    trop longue cette fonction est mise dans Graphe.cpp
ostream & operator<<(ostream & f, const Graphe &  a);
```

Les methodes des classes

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


class Arc;
class Sommet;

class Sommet { public :
  int numero;                                          //    le numero du sommet
  int couleur;                                        //    la couleur du sommet
  int nbarcs;                                   //    nb d'arc contenant ce sommet

  Arc ** a;                                //    listes des arcs contenant ce sommet
                                                    //    constructeur par default
  Sommet(int n=0) : numero(n),couleur(0),nbarcs(0),a(0) {}
  void Set(int i) {numero=i;}
  Sommet & operator=(int i) { numero=i;return *this;}
  int o() const {return nbarcs;}

                        //    pas d'operateurs de copies, c'est une class simple
                        //    dans le cas 2 on interdit la copie de sommet
  ~Sommet(){delete a;}
private:
  Sommet(const  Sommet &);
  void operator =(const  Sommet &);
};
```

```
class Arc {public:
  int numero;                                          //    le numero de l'arc
  Sommet *s[2];                    //    un tableau de 2 pointeurs sur les sommets
                                                       //   ( ici il faut refechir )

  Arc(): numero(0) {s[0]=s[1]=0; }                //    un constructeur par defaut
                                                     //   pour faire de tableau arc

                                                                //    les methodes ...
  const Sommet & operator[](int i) const
  { assert(i >=0 && i <=1); return (*s[i]);}

  Sommet & operator[](int i)
  { assert(i >=0 && i <=1); return (*s[i]);}


                                                             //    le vrai constructeur.
  void Set(int i,Sommet & a, Sommet & b) {numero=i;s[0]=&a; s[1]=&b;}

  const Sommet & SommetOppose(Sommet & si) const
  { assert(&si == s[0] || &si == s[1]);
  return  *(&si == s[0]? s[1] : s[0]); }

  Sommet & SommetOppose(Sommet & si)
  { assert(&si == s[0] || &si == s[1]);
  return  *(&si == s[0]? s[1] : s[0]); }


  Sommet * SommetOppose(Sommet * si) const
  { assert(si == s[0] || si == s[1]);
  return  (si == s[0]? s[1] : s[0]); }

private:
                                 //    pas de copie on a toujours l'arc d'un graphee
  Arc(const  Arc & );
  void operator =(const  Arc & );
};
```

```
class Graphe { public:
  int n;                                                 //    n nombre de sommets
  int m;                                                    //    m nombre d'arcs
  Sommet * s;                                          //    le tableau des sommets
  Arc * a;                                               //    le tableau des Arcs


  Arc & operator[](int i){ return a[CheckA(i)];}
  const Arc & operator[](int i) const { return a[CheckA(i)];}

  Sommet & operator()(int i){ return s[CheckS(i)];}
  const Sommet & operator()(int i) const { return s[CheckS(i)];}
                                    //    G[i] est l'arc i et G(i) est le sommet i
```

```
  int CheckA(int i) const { assert(i>=0 && i < m); return i;}
  int CheckS(int i) const { assert(i>=0 && i < n); return i;}

  void MiseAJour();
  Graphe(int  );                                          //    un graphe c
  Graphe(const char * filename);                 //    un graphe lue dans filename
  ~Graphe()
  {
    cout << " delete Graphe: nb sommets " << n << " , nb arcs  " << m << endl;
    delete [] s;
    delete [] a;
  }

                                                    //    pas operateur de copie
private:
  Graphe(const Graphe &);
  Graphe & operator=(const Graphe &);
};



inline  ostream & operator<<(ostream & f, const Sommet &  s)
{
  if(&s) f << " s " << s.numero << " c " << s.couleur << " ";
  else cout << " s NULL ";
  return f;
}

inline ostream & operator<<(ostream & f, const Arc &  a)
{ f << " a " << a.numero << " :   " << a[0].numero << " "
    << a[1].numero << " "; return f;}


                        //    trop longue cette fonction est mise dans Graphe.cpp
ostream & operator<<(ostream & f, const Graphe &  a);
```

L'agorithme qui colorie la composante connexe du sommet s non colorie. remarque Ici, les couleurs sont simplement des entiers et la «non couleur» est 0

```
 void Coloriage(const Graphe & G,Sommet & s,int couleur)
{
  kpile++;                                          //    incrementation de la pile
  mpile = max(kpile,mpile);                    //    stockage de la profondeur max
  assert(couleur);
  s.couleur = couleur;
  for (int k=0;  k <s.nbarcs;k++)
    {
      Sommet & sv = s.a[k]->SommetOppose(s);          //    le sommet oppose a s
      if  (!sv.couleur ) Coloriage(G,sv,couleur);
    }
  kpile--;                                          //    decrementation de pile
}
```

Le programme principale avec impression :

```
#include "Graphe-v3.hpp"
#include <fstream>
          //    ruse pour calcule la profondeur maximal de pile d'appele recursif
                                              //    on utilise des variables global.
int kpile=0;                                                  //    profondeur courante
int mpile =0;                                                  //    profondeur maximal

int main(int argc,char ** argv)
{
  const char * filename = argc>1 ? argv[1] : "graphe.txt";
  Graphe G(filename);

                                             //    si le graphe est petit on affiche
  if (G.m < 100)
  cout << " " << filename << " " <<  G << endl;


  int couleur=0;
  for (int s=0;s<G.n;++s)
    G.s[s].couleur = 0;

  for (int s=0;s<G.n;++s)
    if ( G.s[s].couleur == 0)
      {
        couleur++;
        cout << " Nouvelle composante connexe sommet depart s :  " << s
             << "  Nu = " << couleur << endl;
        Coloriage(G,G.s[s],couleur);
      }
```

Remarque : la complexité de l'algorithme est en $o(n+2m)$. Effectivement on a vue, 1 fois les sommet et deux fois les arcs. Par contre la profondeur de recurcivite est énorme donc cet algoritme ne marche moralemenent pas. Pour les faire fonctionner, il suffit de «derécurciver» l'algorithme en gérant la pile à la main comme dans la fonction `Coloriage_sans_recussion`.

```
  cout << " Nb de composantes connexe du Graphe " << couleur << endl;
  cout << " profondeur max de pile = " << mpile << endl;
}
```

Pour les fanatiques, voilà la version non récursive de l'algorithmes.

```
int  Coloriage_sans_recussion(Graphe & G,Sommet * s,int couleur)
{
                                                    //    Attention le pointeur s
  int sizepile=G.n;
  Sommet  ** ps = new Sommet*[sizepile];
  int * pk = new int[sizepile];
  int niveau=-1,niveauMax=0;
  int nbcolorier=0;
```

```
 L10:                                //    ici: point entrée de la routine recursive
  niveau++;                                        //    change le niveau de la pile
  assert(niveau < sizepile);                                    //    vérification
  ps[niveau]=s;                                               //    sommet courant
  pk[niveau]=0;                                         //    premiere arc du sommet
  niveauMax=max(niveauMax,niveau);
  s->couleur = couleur;                                       //    colorie le sommet
  nbcolorier++;                  //    pour calculer le nombre de sommets colorier.

  while (niveau>=0)                               //    tant que la pile est non vide
    {
      Sommet * sn=ps[niveau];                                 //    le sommet courant
      while ( (pk[niveau]) <sn->nbarcs)                       //    si il y a des arcs
        {
          s = sn->a[pk[niveau]++]->SommetOppose(sn);   //    le sommet oppose a s
          if  (!s->couleur ) goto L10;          //    appel avec le nouveau sommet
        }
      --niveau;  //    on a fin le sommet sn et le niveau courant, donc on depile
    }

  delete [] ps;
  delete [] pk;
  cout << "   profondeur de pile = " << niveauMax << endl;
  return nbcolorier;
}
```

7.6 Seconde Implémentation

Bien sur, nous aurions pu utiliser, la construction de l'image réciproque pour construite les listes des arcs de sommets donnée. Pour cela, il suffit de remarquer que la listes des arcs de sommets données est l'inversion des deux fonctions $F_0 : a \mapsto$ `G.a[a][0].numero` et $F_1 : a \mapsto$ `G.a[a][1].numero`, en faite on peut faire un optimisation en remarquant $\{0, ..., m-1\} \times \{0, 1\}$ est en bijection avec $\{0, ..., 2n-1\}$ avec l'application $(a, i) :\mapsto p = 2a + i$ et que son l'application inverse est $p \mapsto (p/2, p \, mod \, 2)$. Le code pour construire, les listes est :

```
  int fin = -1;                          //    ce n'est pas un numero d'arête valide
  int * depart = new int[G.n];
  int * suivant = new int[G.m*2];
  for (int k = 0; k < G.n;k++)
    depart[k]=fin;
  for (int p=0;p<G.m*2;p++)
    {
      int k = G[p/2][p%2].numero;
      suivant[p]=depart[k];              //    met la liste ancienne dans la case p
      depart[k] = p;                                  //    on met le départ pour p
    }
```

7.7 Maillage et Triangulation 2D

7.8 Les classes de Maillages

Nous allons définir les outils informatiques en C++ pour utiliser des maillages, pour cela nous utilisons la classe R2 définie au paragraphe 5.1 page 54. Le maillage est formé de triangles qui sont définis par leurs trois sommets. Mais attention, il faut numéroter les sommets. La question classique est donc de définir un triangle : soit comme trois numéro de sommets, ou soit comme trois pointeurs sur des sommets (nous ne pouvons pas définir un triangle comme trois références sur des sommets car il est impossible initialiser des références par défaut). Les deux sont possibles, mais les pointeurs sont plus efficaces pour faire des calculs, d'où le choix de trois pointeurs pour définir un triangle. Maintenant les sommets seront stockés dans un tableau donc il est inutile de stocker le numéro du sommet dans la classe qui définit un sommet, nous ferons une différence de pointeur pour retrouver le numéro d'un sommet, ou du triangle du maillage.
Un maillage (classe de type Mesh) contiendra donc un tableau de triangles (classe de type Triangle) et un tableau de sommets (classe de type Vertex2), bien sur le nombre de triangles (nt) , le nombre de sommets (nv), de plus il me paraît naturel de voir un maillage comme un tableau de triangles et un triangle comme un tableau de 3 sommets.
Remarque : Les sources de ces classes et méthodes sont disponible dans l'archive
http://www.ann.jussieu.fr/~hecht/ftp/cpp/EF.tar.bz2
avec un petit exemple.

7.8.1 La classe `Label`

Nous avons vu que le moyen le plus simple de distinguer les sommets appartenant à une frontière était de leur attribuer un numéro logique ou étiquette (*label* en anglais). Rappelons que dans le format FreeFem++ les sommets intérieurs sont identifiés par un numéro logique nul.
De manière similaire, les numéros logiques des triangles nous permettront de séparer des sous-domaines Ω_i, correspondant, par exemple, à des types de matériaux avec des propriétés physiques différentes.
La classe Label va contenir une seule donnée (lab), de type entier, qui sera le numéro logique.

Listing 7: *(label.hpp - la classe Label)*

```
 class Label {
  friend ostream& operator <<(ostream& f,const Label & r  )
  { f <<  r.lab; return f; }
  friend istream& operator >>(istream& f, Label & r  )
  { f >>  r.lab; return f; }
public:
  int lab;
  Label(int r=0):lab(r){}
  int onGamma() const { return lab;}
};
```

Cette classe n'est pas utilisée directement, mais elle servira dans la construction des classes pour les sommets et les triangles. Il est juste possible de lire, écrire un label, et tester si elle est nulle.

Listing 8: *(utilisation de la classe Label)*

```
Label r;
cout << r;                                          //    écrit r.lab
cin  >> r;                                          //    lit r.lab
if(r.onGamma()) { ..... }                  //    à faire si la r.lab != 0
```

7.8.2 La classe `Vertex2` (modélisation des sommets 2d)

Il est maintenant naturel de définir un sommet comme un point de $\mathbb{R}^2$ et un numéro logique `Label`. Par conséquent, la classe `Vertex2` va dériver des classes `R2` et `Label` tout en héritant leurs données membres et méthodes :

Listing 9: *(Mesh2d.hpp - la classe Vertex2)*

```
class Vertex2 : public R2,public Label {
  friend inline ostream& operator <<(ostream& f, const Vertex2 & v )
    { f << (R2) v << ' ' << (Label &) v  ; return f; }
  friend inline istream& operator >> (istream& f,  Vertex2 & v )
      { f >> (R2 &) v >> (Label &) v; return f; }
   public:
   Vertex2() : R2(),Label(){};
   Vertex2(R2 P,int r=0): R2(P),Label(r){}
   private:                          //    pas de copie pour ne pas prendre l'adresse
     Vertex2(const Vertex2 &);
     void operator=(const Vertex2 &);
};
```

Nous pouvons utiliser la classe `Vertex2` pour effectuer les opérations suivantes :

Listing 10: *(utilisation de la classe Vertex2)*

```
Vertex2 V,W;                                  //    construction des sommets V et W
cout << V ;                                          //    écrit V.x, V.y , V.lab
cin  >> V;                                           //    lit V.x, V.y , V.lab
R2  O = V;                                           //    copie d'un sommet
R2  M = ( (R2) V +  (R2) W ) *0.5;        //    un sommet vu comme un point de R2
(Label) V                    //    la partie label d'un sommet (opérateur de cast)
if (!V.onGamma()) { .... }     //    si V.lab = 0, pas de conditions aux limites
```

Remarque 4. *Les trois champs* `(x,y,lab)` *d'un sommet sont initialisés par* `(0.,0.,0)` *par défaut, car les constructeurs sans paramètres des classes de base sont appelés dans ce cas.*

7.8.3 La classe `Triangle` (modélisation des triangles)

Un triangle sera construit comme un tableau de trois pointeurs sur des sommets, plus un numéro logique (*label*). Nous rappelons que l'ordre des sommets dans la numérotation locale ($\{0,1,2\}$) suit le sens trigonométrique. La classe `Triangle` contiendra également une donnée supplémentaire, l'aire du triangle (`area`), et plusieurs fonctions utiles :

- `Edge(i)` qui calcule le vecteur «arête du triangle» opposée au sommet local i ;
- `H(i)` qui calcule directement le gradient de la i coordonnée barycentrique λ^i par la formule :

$$\nabla\lambda^i|_K = H_K^i = \frac{(q^j - q^k)^\perp}{2\,aire_K} \tag{7}$$

où l'opérateur $\perp$ de $\mathbb{R}^2$ est défini comme la rotation de $\pi/2$, ie. $(a,b)^\perp = (-b,a)$, et où les q^i, q^j, q^k sont les coordonnées des 3 sommets du triangle.

Listing 11: *(Mesh2d.hpp - la classe Triangle)*

```
class Triangle: public Label {
                                                          //    variable prive
  Vertex2 *vertices[3];                          //    an array of 3 pointer to vertex
  public:
  R area;
  Triangle() :area() {};                              //    constructor empty for array
  Vertex2 & operator[](int i) const {
     ASSERTION(i>=0 && i <3);
     return *vertices[i];}                  //    to see triangle as a array of vertex
  void init(Vertex2 * v0,int i0,int i1,int i2,int r)
    { vertices[0]=v0+i0; vertices[1]=v0+i1; vertices[2]=v0+i2;
      R2 AB(*vertices[0],*vertices[1]);
      R2 AC(*vertices[0],*vertices[2]);
      area = (AB^AC)*0.5;
      lab=r;
      ASSERTION(area>0);
    }
  R2 Edge(int i) const {ASSERTION(i>=0 && i <3);
       return R2(*vertices[(i+1)%3],*vertices[(i+2)%3]);}      //    opposite edge
vertex i
  R2 H(int i) const { ASSERTION(i>=0 && i <3);
       R2 E=Edge(i);return E.perp()/(2*area);}                          //    heigth

  void Gradlambda(R2 * GradL) const
  {
    GradL[1]= H(1);
    GradL[2]= H(2);
    GradL[0]=-GradL[1]-GradL[2];
```

```
  }

  R lenEdge(int i) const {ASSERTION(i>=0 && i <3);
      R2 E=Edge(i);return sqrt((E,E));}

                                                    //    Transformation: K̂ ↦ K
  R2 operator()(const R2 & Phat) const   {
    const R2 &A =*vertices[0];
    const R2 &B =*vertices[1];
    const R2 &C =*vertices[2];
    return (1-Phat.x- Phat.y)* A + Phat.x *B  +Phat.y*C ;}

 private:
   Triangle(const Triangle &);                //    pas de construction par copie
   void operator=(const Triangle &);          //    pas affectation par copy
 public:                  //   -- Ajoute 2007 pour un ecriture generique 2d 3d --
  R  mesure() const {return area;}
  static const int nv=3;
};
```

Remarque 5. *La construction effective du triangle n'est pas réalisée par un constructeur, mais par la fonction `init`. Cette fonction est appelée une seule fois pour chaque triangle, au moment de la lecture du maillage (voir plus bas la classe `Mesh`).*

Remarque 6. *Les opérateurs d'entrée-sortie ne sont pas définis, car il y a des pointeurs dans la classe qui ne sont pas alloués dans cette classe, et qui ne sont que des liens sur les trois sommets du triangle (voir également la classe `Mesh`).*

Regardons maintenant comment utiliser cette classe :

Listing 12: *(utilisation de la classe Triangle)*

```
                                  //    soit T un triangle de sommets A, B, C ∈ ℝ²
                                     //    ------------------------------------
  Triangle::nv;                    //    nombre de sommets d'un triangle (ici 3)
  const Vertex2 & V =  T[i];                  //    le sommet i de T (i ∈ {0,1,2}
  double a = T.area;                                    //    l'aire de T
  R2 AB = T.Edge(2);                        //    "vecteur arête" opposé au sommet 2
  R2 hC = T.H(2);       //    gradient de la fonction de base associé au sommet 2
  R l = T.lenEdge(i);               //    longueur de l'arête opposée au sommet i
  (Label) T  ;                                  //    la référence du triangle T
  R2 G(T(R2(1./3,1./3)));                         //    le barycentre de T
  Triangle T;
  T.init(v,ia,ib,ic,lab);     //    initialisation du triangle T avec les sommets
                                     //    v[ia],v[ib],v[ic] et l'étiquette lab
                                           //    (v est le tableau des sommets)
```

7.8.4 La classe `Seg` (modélisation des segments de bord)

On fait la même type de classe que les triangles mais juste avec deux sommets, on utilisera cette classe pour calculer les intégrales pour les conditions aux limites de type Neumann,

Listing 13: *(Mesh2d.hpp - la classe Seg)*

```
class Seg: public Label {
                                                          //    variable prive
  Vertex2 *vertices[2];                          //    an array of 3 pointer to vertex
public:
  R l;                                                      //    longueur du segment
  Seg() :l() {};                                  //    constructor empty for array
  Vertex2 & operator[](int i) const {
    ASSERTION(i>=0 && i <2);
    return *vertices[i];}                    //    to see traingle as a array of vertex
  void init(Vertex2 * v0,int * iv,int r)
  {
    vertices[0]=v0+iv[0];
    vertices[1]=v0+iv[1];
    R2 AB(*vertices[0],*vertices[1]);
    l=  AB.norme();
    lab=r;
    ASSERTION(l>0);
  }


                                                        //    Transformation: [0,1] ↦ K
  R2 operator()(const R & Phat) const    {
    const R2 &A =*vertices[0];
    const R2 &B =*vertices[1];
    return (1-Phat)* A + Phat *B ;}

private:
  Seg(const Seg &);                              //    pas de construction par copie
  void operator=(const Seg &);                   //    pas affectation par copy
public:
  R  mesure() const {return l;}
  static const int nv=2;
};
```

Regardons maintenant comment utiliser cette classe :

Listing 14: *(utilisation de la classe Seg)*

```
                                                  //    soit K un Seg de sommets A, B ∈ ℝ²
                                                //    ----------------------------------
  Seg::nv;                                      //    nombre de sommets d'un Seg (ici 3)
  const Vertex2 & V =  K[i];                          //    le sommet i de T (i ∈ {0,1}
```

```
  double a = K.l;                                          //    la longueur de K
  (Label) K  ;                                          //    le label du triangle Seg
  R2 G(T(0.5)));                                           //    le barycentre de K
  Seg K;
  K.init(v,ia,ib,lab);              //    initialisation d'un Seg avec les sommets
                                        //    v[ia],v[ib] et l'étiquette lab
                                        //    (v est le tableau des sommets)
```

7.8.5 La classe `Mesh2` (modélisation d'un maillage 2d)

Nous présentons, pour finir, la classe maillage (`Mesh`) qui contient donc :

- le nombre de sommets (`nv`), le nombre de triangles, le nombre de arêtes de bord (`nbe`) (`nt`) ;
- le tableau des sommets ;
- le tableau des triangles ;
- le tableau des arêtes du bord ;

Listing 15: *(Mesh2d.hpp - la classe Mesh)*

```
class Mesh2
{
public:
  typedef Triangle Element;
  typedef Seg  BorderElement;
  typedef R2 Rd;
  typedef Vertex2  Vertex;
  int nv,nt, nbe;
  R area,peri;
  Vertex2 *vertices;
  Triangle *triangles;
  Seg * borderelements;
  Triangle & operator[](int i) const {return triangles[CheckT(i)];}
  Vertex2 & operator()(int i) const {return vertices[CheckV(i)];}
  Seg & be(int i) const {return borderelements[CheckBE(i)];}
  Mesh2(const char * filename);                                  //    read on a file
                                                              //    to get numbering:
  int operator()(const Triangle & t) const {return CheckT(&t - triangles);}
  int operator()(const Triangle * t) const {return CheckT(t - triangles);}
  int operator()(const Vertex2 & v) const {return CheckV(&v - vertices);}
  int operator()(const Vertex2  * v) const{return CheckV(v - vertices);}
  int operator()(const Seg & v) const {return CheckBE(&v - borderelements);}
  int operator()(const Seg  * v) const{return CheckBE(v - borderelements);}

  //    the global Number of vertex j of triangle it (j=0,1,2, it=0, .., nt-1)
  int operator()(int it,int j) const {return (*this)(triangles[it][j]);}
                                                              //    to check the bound
  int CheckV(int i) const { ASSERTION(i>=0 && i < nv); return i;}
  int CheckT(int i) const { ASSERTION(i>=0 && i < nt); return i;}
  int CheckBE(int i) const { ASSERTION(i>=0 && i < nbe); return i;}
  ~Mesh2() { delete [] vertices; delete [] triangles;delete [] borderelements; }
 private:
```

```
   Mesh2(const Mesh2 &);                    //    pas de construction par copie
   void operator=(const Mesh2 &);           //    pas affectation par copy
};
```

Avant de voir comment utiliser cette classe, quelques détails techniques nécessitent plus d'explications :

- Pour utiliser les opérateurs qui retournent un numéro, il est fondamental que leur argument soit un pointeur ou une référence ; sinon, les adresses des objets seront perdues et il ne sera plus possible de retrouver le numéro du sommet qui est donné par l'adresse mémoire.
- Les tableaux d'une classe sont initialisés par le constructeur par défaut qui est le constructeur sans paramètres. Ici, le constructeur par défaut d'un triangle ne fait rien, mais les constructeurs par défaut des classes de base (ici les classes `Label, Vertex2`) sont appelés. Par conséquent, les labels de tous les triangles sont initialisées et les trois champs `(x,y,lab)` des sommets sont initialisés par `(0.,0.,0)`. Par contre, les pointeurs sur sommets sont indéfinis (tout comme l'aire du triangle).

Tous les problèmes d'initialisation sont résolus une fois que le constructeur avec arguments est appelé. Ce constructeur va lire le fichier `.msh` contenant la triangulation.

Listing 16: *(Mesh2d.cpp - constructeur de la classe Mesh)*

```
#include <cassert>
#include <fstream>
#include <iostream>
#include "ufunction.hpp"
#include "Mesh2d.hpp"

Mesh2::Mesh2(const char * filename)
  : nv(0),nt(0),nbe(0),
    area(0),peri(0),
    vertices(0),triangles(0),borderelements(0)
{                                                          //    read the mesh
  int i,iv[3],ir;
  ifstream f(filename);
  if(!f) {cerr << "Mesh2::Mesh2 Erreur openning " << filename<<endl;exit(1);}
  cout << " Read On file \\"" <<filename<<"\\""<<  endl;
  f >> nv >> nt >> nbe;
  cout << " Nb of Vertex " << nv << " Nb of Triangles " << nt
       << " Nb of Border Seg : " << nbe << endl;
  assert(f.good() && nt && nv );
  triangles = new Triangle [nt];
  vertices = new Vertex[nv];
  borderelements = new Seg[nbe];
  area=0;
  assert(triangles && vertices);
  for (i=0;i<nv;i++)
    {
      f >> vertices[i];
```

```
    assert(f.good());
  }
for (i=0;i<nt;i++)
  {
    f >> iv[0] >> iv[1] >> iv[2] >> ir;
    assert(f.good() && iv[0]>0 && iv[0]<=nv && iv[1]>0
           && iv[1]<=nv && iv[2]>0 && iv[2]<=nv);
    for (int v=0;v<3;++v) iv[v]--;
    triangles[i].init(vertices,iv,ir);
    area += triangles[i].area;
  }
for (i=0;i<nbe;i++)
  {
    f >> iv[0] >> iv[1]  >> ir;
    assert(f.good() && iv[0]>0 && iv[0]<=nv && iv[1]>0 && iv[1]<=nv);
    for (int v=0;v<2;++v) iv[v]--;
    borderelements[i].init(vertices,iv,ir);
    peri += borderelements[i].l;
  }

cout << " End of read: area = " << area << "  perimeter: " << peri << endl;
}
```

L'utilisation de la classe `Mesh` pour gérer les sommets, les triangles devient maintenant très simple et intuitive.

Listing 17: *(utilisation de la classe Mesh2)*

```
Mesh2 Th("filename");            //    lit le maillage Th du fichier "filename"
Th.nt;                                                //    nombre de triangles
Th.nv;                                                  //    nombre de sommets
Th.neb;                                            //    nombre de éléments de bord
Th.area;                                            //    aire du domaine de calcul
Th.peri;                                         //    perimetre du domaine de calcul

Triangle & K = Th[i];                                  //    triangle i , int i∈ [0,nt[
R2 A=K[0];                                 //    coordonnée du sommet 0 sur triangle K
R2 G=K(R2(1./3,1./3)):                                 //    le barycentre de K.
R2 DLambda[3];
K.Gradlambda(DLambda);                    //    calcul des trois ∇λ_i^K pour i = 0,1,2
Vertex2   & V = Th(j);                                 //    sommet j , int j∈ [0,nv[
Seg &   BE=th.be(l);                                 //    Seg du bord, int l∈ [0,nbe[
R2 B=BE[1];                                //    coordonnée du sommet 1 sur Seg BE
R2 M=BE(0.5);                                          //    le milieu de BE.
int j = Th(i,k);        //    numéro global du sommet k∈ [0,3[ du triangle i∈ [0,nt[
Vertex2 & W=Th[i][k];      //    référence du sommet k∈ [0,3[ du triangle i∈ [0,nt[

int ii = Th(K);                                        //    numéro du triangle K
int jj = Th(V);                                          //    numéro du sommet V
int ll = Th(BE);                                    //    numéro de Seg de bord BE

assert( i == ii && j == jj);                                  //    vérification
```

8 Utilisation de la STL

8.1 Introduction

La STL un moyen pour unifier utilisation des différents type de stockage informatiques que sont les, tableau, liste, arbre binaire, etc....
Ces moyens de stockage sont appelés des conteneurs. A chaque type de conteneur est associé un `iterator` et `const_iterator` , qui une formalisation des pointeurs de parcours associés au conteneur.
Si T est un type conteneur de la STL , alors pour parcourir sans modification les éléments du conteneur l de type T, il suffit d'écrire

```
for (typename T::const_iterator i=l.begin(); i != l.end(); ++i)
  {
   *i;                          //    sera la valeur associe a vos données stoker.
cout << * i << endl;       //    imprime une valeur par ligne de votre conteneur.
  }
```

Pour parcourir avec modification possible les éléments du conteneur l de type T, il suffit d'écrire

```
for (typename T::iterator i=l.begin(); i != l.end(); ++i)
  {
   *i;                          //    sera la valeur associe a vos données stoker.
cout << * i << endl;       //    imprime une valeur par ligne de votre conteneur.
  }
```

Les conteneurs utiles pour stoker un jeu de valeur de type `V`.

vector Pour stocker, un tableau v de valeurs `{vector<V> v;`, et la ligne d'include associé est `#include <vector>`.

list Pour stoker une liste doublement chainées de valeur, `list<V> l;`, et la ligne d'include associé est `#include <list>`.

map Pour stoker des pairs de (clef de type `K`, valeur) automatiquement trié par rapport aux clefs, avec unicité des clefs dans le conteneurs. Attentions ici les valeurs stokes seront donc des `pair<K,V>`.

multimap Pour stoker des pairs de (clef de type `K`, valeur) automatiquement trié par rapport aux clefs, avec non unicité des clefs dans le conteneurs. Attentions ici les valeurs stokes seront donc des `pair<K,V>`.

set Pour stoker un ensemble de valeurs ordonnées, c'est un `map` sans valeur.

unordered_map come les maps mais sans ordre base sur les méthodes de hachage

```
<algorithm>   copy(),find(),sort()
<array>      array
<chrono>     duration,time_point
<cmath>       sqrt(),pow()  complex,sqrt(),pow()
<complex>
<fstream>    fstream,ifstream,ofstream
```

```
<iostream>  istream,ostream,cin,cout

<map>
<memory>     unique_ptr,shared_ptr,allocator
<random>
<regex>      regex, smatch
<string>
<set>
<sstream>
<thread>
<unordered_map>  unordered_map,unordered_multimap
<utility>        move(),swap(),pair
<vector>
```

8.2 exemple

Le répertoire des sources : http://www.ann.jussieu.fr/~hecht/ftp/cpp/stl4

```
                //    voila un exemple assez complet des possibilites de la STL
                 //    ---------------------------------------------------------
#include <list>
#include <vector>
#include <map>
#include <set>
#include <queue>
#include <iostream>
#include <fstream>
#include <string>
#include <cassert>
#include <algorithm>
#include <iterator>
#include <complex>

using namespace std;


                                                    //    pour afficher une pair
template<typename A,typename B>
ostream & operator<<(ostream & f, pair<A,B> ab)
{
    f << ab.first << " : " <<  ab.second;
    return f;
}

                                  //    ------------------------------------//
                                   //    pour afficher un container de la stl //
                                  //    ------------------------------------//
template<typename T>
void  show(const char * s,const T & l,const char * separateur="\\n")
{
  cout << s <<  separateur;
  for (typename T::const_iterator i=l.begin(); i != l.end(); ++i)
```

```
    cout << '\\t' << * i <<  separateur;
  cout << endl;
}

                //    ------------------------------------------------------ //
                  //    pour afficher un container de la stl dans l'autre sens //
                //    ------------------------------------------------------ //
template<typename T>
void  showr(const char * s,const T & l,const char * separateur="\\n")
{
  cout << s <<  separateur;
  for (typename T::const_reverse_iterator i=l.rbegin(); i != l.rend(); ++i)
                                //    for ( auto i=l.rbegin(); i != l.rend(); ++i)
    cout << '\\t' << * i << * separateur;
  cout << endl;
}
                                    //    ---------------------------------------
/*  ordre lexicographie sur les pair est deja dans la stl */
                                      //    -------------------------------------
/*  donc inutile
  template<typename A,typename B>
  bool operator<(const pair<A,B> &a,const pair<A,B> & b) {
    return a.first == b.first ? ( a.second < b.second ): a.first < b.first;
  }
*/

int main() {
                                     //    -------------------------------------
                                      //    les vecteurs sont des tableaux ----
                                     //   -------------------------------------
  {
    vector<double> v;                                       //    un vector vide
    v.push_back(1.);
    v.push_back(-1.);
    v.push_back(-5.);
    v.push_back(4.);
                                //    v.push_front(-9); not implemented in vector


                                       //    v.sort(); // n'est pas implemente
    v[1]++;
    vector<double>::iterator k=v.begin()+3;
    v.erase(v.begin()+3);
    v.insert(v.end()-1,1000.);          //    insert 1000 en avant avant dernier
           //    v.erase(find(v.begin(),v.end(),-5.)); // ok retire la valeur -5
    show("vector      :",v,"  ");
    showr("vector (reverse)       :",v,"  ");
    sort(v.begin(),v.end());                                //    trie le vector
    show("vector trie :",v,"  ");
    vector<double> v10(10);                       //    un vecteur de 10 elements
    v10[1]= 3;
      //    remarque dans la STL, le programe suivant n'est pas en o(n^2) operation
    int n = 1000000;
    for (int i=0;i<n;++i)
      v.push_back(i);
     //    mais en o(nlog(n)) car la STL reserve de la place en plus dans le tableau
                                        //    de l'ordre d'une fraction de n
```

```
  }


                        //    ------------------------------------------------
                        //    ----- Test des listes doublement chainees----------
                        //   ------------------------------------------------
  {
    list<double> l;
    l.push_back(1.);
    l.push_back(5.);
    l.push_back(3.);
    l.push_back(6.);
    l.push_back(3.);
    list<double>::iterator l5=find(l.begin(),l.end(),5.);
    cout << " taille de la liste : o(n) operation " << l.size() << endl;
    cout << " liste vide ? " << l.empty() << endl;
                                                                  //    remarque:
                          //    list< double>::iterator l3=l.begin()+3; interdit
                       //    car un iterator de list n'est pas aleatoire (random)

    assert( l5 != l.end());
    l.insert(l5,1000.);                                 //    insert 1000. avant l5
    show("list    :",l,"  ");
    list<double> ll(l);                                  //    copie la liste
    show("list ll : ",l,"  ");
    l.splice(l5,ll);  //    deplace la list ll dans l avant l5 (sans copie =>vide
ll)
    cout << " ll.size = " << ll.size() << endl;
    assert( ll.empty());                                    //    la liste est vide
                                                          //    remarque utile:
                      //    empty pour savoir si un liste de vide O(1) operations
                                 //    car size() est en O(size()) operations.
    ll.push_front(-3.);                       //    ajoute un element a la liste ll
    list<double> vv;
    vv.push_back(1.);
    vv.push_back(5.);

    l.insert(l5,vv.begin(),vv.end());                //    insert en copiant avant
    l.insert(l5,ll.begin(),ll.end());                //    insert en copiant avant
    l.erase(l5);                                 //    supprime l'element 5 de la liste
              //    remarque l5 est valide tant que l'element n'est pas supprimer
    {
                                                                              //
const  list<double> & ll(l);             //    un autre list ll (bloque different)
                              //    recherche de la valeur 5 dans l'autre liste ll
list<double>::const_iterator l5=find(ll.begin(),ll.end(),5.);
    }


    cout << " impression with copy : ";
    copy(l.begin(),l.end(),ostream_iterator<double>(cout, " "));
    cout << endl;

      //    sort(l.begin(),l.end());// Erreur car un iterator de liste n'est pas
ramdon
    l.sort();                                          //    trie la liste en n log(n)
    show("list trie : ",l,"  ");
```

```
  l.clear();                                                    //    vide la liste
}

                                        //    ---------------------------------- //
                                              //    exemple d'utilisation des map //
                                        //    ---------------------------------- //

{
  map<string,string> m;
  m["hecht"]="+33 1 44 27 44 11";
  m["toto"]="inconnue";
  m["ddd"]="a supprime";
  m["aaaa"]="inconnueaaa";
  m.erase("ddd");
  map<string,string>::iterator  ff=m.find("toto");
  if (ff!= m.end())
    cout << " find " <<  *ff<< endl;
  else
    cout << " pas trouver " << endl;
  {
    pair<map<string,string>::iterator,bool> pbi
       =m.insert(make_pair<string,string>("toto","inconnuetoto"));
    cout << "insert map : " << *pbi.first << " " << pbi.second << endl;
  }
  {
    pair<map<string,string>::iterator,bool> pbi
       =m.insert(make_pair<string,string>("toto1","inconnuetoto"));
    cout << "insert map : " << *pbi.first << " " << pbi.second << endl;
  }
  show("map",m);
  cout << " -------------------------- \\n";
}

                                                          //    -----------------
                                                              //    un ensemble .
                                                          //    -----------------
{
                            //    ---------------------------------------------
                                      //    un petit exemple d'ensemble d'entier
              //    les elements doivent etre comparable ( operateur "<" existe)
  set<int>   S;
  S.insert(1);
  S.insert(2);
  S.insert(2);
              //    le test apartenence est i\\inS \\texttt(S.find(i) != S.end())
  show("set S :",  S, "  ");
  for (int i=0;i<5;i++)
    if ( S.find(i) != S.end())
cout << i << " est dans l'ensemble S\\n";
    else
cout << i << " n'est pas dans l'ensemble S\\n";
  cout << " -------------------------- \\n";
}


{
  map<complex<double>,int >  cc;                                //    pas de bug ici
```

```
    complex<double> cca=1;
                //    cc[cca]=2; // bug car pas de comparasion < sur des complex
  }



  /*
    Attention dans une map tout depend de l'ordre.
    Voilà une exemple qui ne fait pas ce qui est attendue generalement
  */
  {
    map<const char *,const char *>  dicofaux;       //    ICI L'ORDRE UTILISE EST
l'adresse memoire.
    const char * b="b";
    const char * c="c";
    char  a[2];
    a[0]='a'; a[1]=0;
    const char * d="d";
    dicofaux[a]="1";
    dicofaux[c]="4";
    dicofaux[d]="3";
    dicofaux[b]="2";
    dicofaux["a"]="22";
    show(" dico faux: ",dicofaux,"\\n");
  }


                          //   ------------------------------------------------
                            //    ici modelisation d'un ensemble de arete defini
                            //    par les numeros d'extermités (first et second)
                               //    de plus a chaque arete on associe un entier
                            //    ------------------------------------------------

                          //    ------------------------------------------------
                            //    une exemple d'ensemble d'arete (pair d'entier)
                                                          //    numerote.
                           //    ------------------------------------------------
  {
    map<pair<int,int>,int > edges;
    int nbe =0, nbedup=0;
    for (int i=0;i<10;i++)
      {

pair<int,int> e(i%5,(i+1)%5);
                                  //    insertion de l'arete e avec la valeur nbe;

                       //    macro generation pour le choisir la méthode (bofbof)
#if 1
                                                          //    methode optimiser
   //    remarque insert retourne une pair de (iterator, bool) ( true => n'existe
pas)
if( edges.insert(make_pair(e,i)).second)
  nbe++;                                                  //    ici nouvelle item
else
  nbedup++;                                             //    ici item existe deja
#else
                               //    Autre methode plus simple mais moins efficace
```

```
if( edges.find(e) == edges.end() )
  {
    nbe++;                                                    //    ici nouvelle item
    edges[e]=i;
  }
else
  nbedup++;                                                   //    ici item existe deja
                          //    remarque edges[e] peut etre un nouvel élément dans
      //    dans ce cas, il est initialisé avec T() qui est la valeur par defaut
                    //    de toute classe / type T, ici on a T == int et int() == 0
#endif
      }
    cout << " nbe = " << nbe <<  " nbedup = " << nbedup << endl;
                     //    du code pour voir les items qui sont ou non dans la map
    for (int i=0;i<10;i++)
{
  pair<int,int> e(i,(i+1));
  if (edges.find(e)!=edges.end() )
    cout << "     trouver   arete  (" << e << ")  \\n";
  else
    cout << " non trouver   arete  (" << e << ") \\n";
}
    show(" les aretes ", edges);
  }
                           //    exemple d'utilisation d'une multi map ( oct 2010)
            //    pour invertion d'une function a valeur entiere non injective ..
        //    peut etre utile car ici l'ensemble d'arrive dans un intervalle tres
grand 2^31 - 1
  {
    int n= 10;
    multimap<int,int> F1;
    typedef multimap< int,int>::iterator  MI;
                                                              //    creation de la multi map
    for(int i=0;i<n;++i)
      {
int Fi =  i*i%25;
F1.insert(make_pair(Fi,i));
      }
    show(" F1 ",F1);
    for (int j=0;j<25;++j)
      {
pair<MI,MI> cj =F1.equal_range(j);
cout <<" F^-1("<< j << " ) = :";
for ( MI k=cj.first; k != cj.second; ++k)
  cout << k->second << " ";
cout << endl;
      }
  }

                                                    //    ---------------------------- //
                                                     //    exemple de queue prioritaire //
                                                    //    ---------------------------- //
  {
    cout << " queue prioritaire " <<endl;
    priority_queue<int> pq;
                                                       //    pour ajoute des valeurs
    pq.push(10);
```

```
    pq.push(1);
    pq.push(5);
                                                    //    pour voir la haut de la queue
      while(!pq.empty())                                //    la queue est telle vide
{
  int t=pq.top();
                                                 //    pour supprimer le haut de la queue
  cout << "top   t = " <<  t << " and pop " <<endl;
  pq.pop();
  if (t== 5) { pq.push(6); cout << " push 6 n"; }   //    je mets 6 dans la queue

}
                    //    il est impossible de parcourir les elements d'une queue
                                                              //    ils sont caches
  }

    return  0;
}
```

8.3 Chaîne de caractères

A FAIRE

8.4 Entrée Sortie en mémoire

Voilà, un dernier truc, il est souvent bien utile de générer de nom de fichier qui dépend de valeur de variable, afin par exemple de numeroter les fichier. Les classes `istringstream` et `ostringstream` permet de faire de entrée sortir en mémoire permette de résoudre se type de problème.

```
#include <sstream>
#include <iostream>
using namespace std;
int main(int argc,char ** argv)
{
  string str;
  char * chaine = "1.2+ dqsdqsd q ";
                                                    //    pour construire la chaine str
  for (char * c=chaine; *c; ++c)
    str+= *c;                                                          //    ajoute

  cout << str << endl;
  istringstream f(str);              //    pour lire dans une chaine de caractere
  double a;
  char c;
  f >> a;
  c=f.get();
  cout << " a= " <<a << " " << c <<  endl;

          //    util pour generer des noms de fichier qui dependent de variables
  ostreamstream ff;                 //    pour ecrire dans une chaine de caracteres
  ff << "toto"<< i << ".txt" << ends;
  cout << " la string associer : " << ff.str() << endl << endl;
  cout << " la C chaine (char *) " << ff.str().c_str() << endl;
```

```
  return 0;
}
```

9 Construction d'un maillage bidimensionnel

Nous nous proposons de présenter dans ce chapitre les notions théoriques et pratiques nécessaires pour écrire un générateur de maillage (*mailleur*) bidimensionnel de type Delaunay-Voronoï, simple et rapide.

9.1 Bases théoriques

9.1.1 Notations

1. Le segment fermé (respectivement ouvert) d'extrémités a, b de $\mathbb{R}^d$ est noté $[a, b]$ (respectivement $]a, b[$).
2. Un ensemble convexe C est tel que $\forall (a, b) \in \mathcal{C}^2, [a, b] \subset C$.
3. Le convexifié d'un ensemble S de points de $\mathbb{R}^d$ est noté $\mathcal{C}(S)$ est le plus petit convexe contenant S et si l'ensemble est fini (*i.e.* $S = \{x^i, i = 1, .., n\}$) alors nous avons :

$$\mathcal{C}(S) = \left\{ \sum_{i=1}^{n} \lambda_i x^i : \quad \forall (\lambda_i)_{i=1,..,n} \in \mathbb{R}_+^n, \text{ tel que } \sum_{i=1}^{n} \lambda_i = 1 \right\}$$

   .
4. Un ouvert Ω est polygonal si le bord $\partial\Omega$ de cet ouvert est formé d'un nombre fini de segments.
5. L'adhérence de l'ensemble O est notée $\overline{O}$.
6. L'intérieur de l'ensemble F est noté $\overset{\circ}{F}$.
7. un k-simplex $(x^0, ..., x^k)$ est le convexifié des $k+1$ points de $\mathbb{R}^d$ affine indépendant (donc $k \leq d$) ,
   — une arête ou un segment est un 1-simplex,
   — un triangle est un 2-simplexe,
   — un trétraèdre est un 3-simplexe ;
   La mesure signée d'un d-simplex $K = (x^0, ..., x^d)$ en dimension d est donnée par

$$mes(x^0, ..., x^d) = \frac{-1^d}{d!} det \begin{vmatrix} x_1^0 & \dots & x_1^d \\ \vdots & \dots & \vdots \\ x_d^0 & \dots & x_d^d \\ 1 & \dots & 1 \end{vmatrix}$$

   et le d-simplex sera dit positif sa mesure est positive.
   les $p - simplexe$ sous d'un $k - simplexe$ sont formés avec $p+1$ points de $(x^0, ..., x^d)$. Les ensembles de $k - simplex$
   — des sommets sera l'ensemble des $k+1$ points (0-simplexe),
   — des arêtes sera l'ensemble des $\frac{(k+1)\times k}{2}$ 1-simplex ,
   — des triangles sera l'ensemble des $\frac{(k+1)\times k\times (k-1)}{6}$ 2-simplex ,

— des hyperfaces sera l'ensemble des $\frac{(k+1)\times k}{2}$ (d-1)-simplex ,
— Les coordonnées barycentriques d'un d-simplexe $K = (x^0, ..., x^d)$ sont des $d+1$ fonctions affines de $\mathbb{R}^d$ dans $\mathbb{R}$ noté $\lambda_i^K, j = 0, ..., d$ et sont défini par

$$\forall x \in \mathbb{R}^d; \qquad x = \sum_{i=0}^{d} \lambda_i^K(x) x^i; \qquad \text{et} \qquad \sum_{i=0}^{d} \lambda_i^K(x) = 1 \tag{8}$$

on a $\lambda_i^K(x^j) = \delta_{ij}$ où δ_{ij} est le symbole de Kroneker. Et les formules de Cramers nous donnent :

$$\lambda_i^K(x) = \frac{mes(x^0, ..., x^{i-1}, x, x^{i+1}, ..., x^d)}{mes(x^0, ..., x^{i-1}, x^i, x^{i+1}, ..., x^d)}$$

8. Le graphe d'une fonction $f : E \mapsto F$ est l'ensemble les points $(x, f(x))$ de $E \times F$.

9.1.2 Introduction

Commençons par définir la notion de maillage simplexial.

Définition 9.1. *Un maillage simplexial $\mathcal{T}_{d,h}$ d'un ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^d$ est un ensemble de $d-$ simplex K^k de $\mathbb{R}^d$ pour $k = 1, N_t$ (triangle si $d = 2$ et tétraèdre si $d = 3$), tel que l'intersection de deux d-simplex distincts $\overline{K}^i, \overline{K}^j$ de $\mathcal{T}_{d,h}$ soit :*
— *l'ensemble vide,*
— *ou p-simplex commun à K et K' avec $p \leq d$*

Le maillage $\mathcal{T}_{d,h}$ couvre l'ouvert défini par :

$$\mathcal{O}_h \overset{def}{=} \overline{\bigcup_{K \in \mathcal{T}_{d,h}}^{\circ} K} \tag{9}$$

De plus, $\mathcal{T}_{0,h}$ désignera l'ensemble des sommets de $\mathcal{T}_{d,h}$ et $\mathcal{T}_{1,h}$ l'ensemble des arêtes de $\mathcal{T}_{d,h}$ et l'ensemble de faces serat $\mathcal{T}_{d-1,h}$. Le bord $\partial\mathcal{T}_{d,h}$ du maillage $\mathcal{T}_{d,h}$ est défini comme l'ensemble des faces qui ont la propriété d'appartenir à un unique d-simplex de $\mathcal{T}_{d,h}$. Par conséquent, $\partial\mathcal{T}_{d,h}$ est un maillage du bord $\partial\mathcal{O}_h$ de $\mathcal{O}_h$. Par abus de langage, nous confondrons une arête d'extrémités (a, b) et le segment ouvert $]a, b[$, ou fermé $[a, b]$.

Remarque 7. *Les triangles sont les composantes connexes de*

$$\mathcal{O}_h \setminus \bigcup_{(a,b) \in \mathcal{T}_{1,h}} [a, b]. \tag{10}$$

Avant toutes choses, les maillages sont une support d'informations, et offre un simple de construire des fonctions continue.

Définition 9.2. *Soit f une fonction à valeur réel défini sur les sommets d'un maillage $\mathcal{T}$, nous allons simplement, définit l'interpolé P_1 lagrange de f note $\mathcal{I}_{\mathcal{T}}(f)$ comme la fonction telle que*

$$\forall K \in \mathcal{T}_d, \ \ \forall x \in K, \qquad \mathcal{I}_{\mathcal{T}}(f) = \sum_{i=0}^{d} \lambda_i^K(q_K^i) \tag{11}$$

où les q_K^i sont les $d+1$ sommets de l'élément K, et les λ_i^K sont coordonnées barycentriques définies en (8).

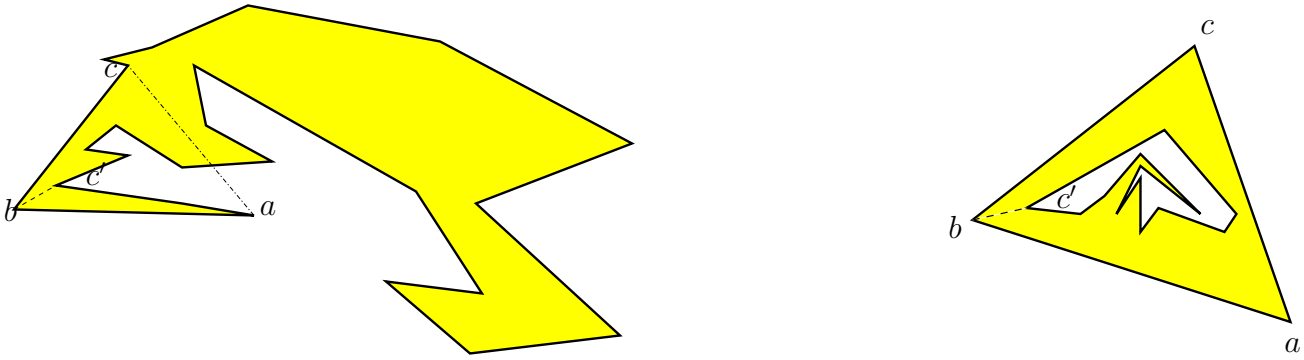
Commençons par donner un théorème fondamental en dimension $d = 2$.

Théorème 9.1. *Pour tout ouvert polygonal $\mathcal{O}_h$ de $\mathbb{R}^2$, il existe un maillage de cet ouvert sans sommet interne.*

L'ensemble S des sommets de ce maillage sont les points anguleux du bord $\partial\mathcal{O}_h$.
La démonstration de ce théorème utilise le lemme suivant :

Lemme 9.3. *Dans un ouvert polygonal $\mathcal{O}$ connexe qui n'est pas un triangle, il existe deux points anguleux α, β tels que $]\alpha, \beta[\subset \mathcal{O}$.*

Preuve : il existe trois points anguleux a, b, c consécutifs du bord $\partial\mathcal{O}$ tel que l'ouvert soit localement du côté droite de $]a, b[$ et tel que l'angle $\widehat{abc} < \pi$, et notons $T = \mathcal{C}(\{a, b, c\})$ le triangle fermé formé de a, b, c et $\overset{\circ}{T}$ le triangle ouvert.



Il y a deux cas :

- $\overset{\circ}{T} \cap S = \emptyset$ alors $]a, c[\subset \mathcal{O}_h$ et il suffit de pendre $]\alpha, \beta[=]a, c[$.
- Sinon, soit $\mathcal{C}$ le convexifié de cette intersection $\overset{\circ}{T} \cap S$. Par construction, ce convexifié $\mathcal{C}$ est inclus dans le triangle ouvert $\overset{\circ}{T}$. Soit le point anguleux c' du bord du convexifié le plus proche de b ; il sera tel que $]b, c'[\subset \mathcal{O}_h$, et il suffit de prendre $]\alpha, \beta[=]b, c'[$.

∎

Preuve du théorème 9.1:
Construisons par récurrence une suite d'ouverts $\mathcal{O}^i, i = 0, ..., k$, avec $\mathcal{O}^0 \overset{def}{=} \mathcal{O}$.
Retirons à l'ouvert $\mathcal{O}^i$ un segment $]a_i, b_i[$ joignant deux sommets a_i, b_i et tel que $]a_i, b_i[\subset \mathcal{O}^i$, tant qu'il existe un tel segment.

$$\mathcal{O}^{i+1} \overset{def}{=} \mathcal{O}^i \setminus]a_i, b_i[\tag{12}$$

Soit N_c le nombre de sommets ; le nombre total de segments joignant ces sommets étant majoré par $N_c \times (N_c - 1)/2$, la suite est donc finie en $k < N_c \times (N_c - 1)/2$.
Pour finir, chaque composante connexe de l'ouvert $\mathcal{O}^k$ est un triangle (sinon le lemme nous permettrait de continuer) et le domaine est découpé en triangles. ∎

Remarque 8. *Malheureusement ce théorème n'est plus vrai en dimension plus grande que 2, car il existe des ouverts polyédriques non-convexe qu'il est impossible de mailler sans point interne.*

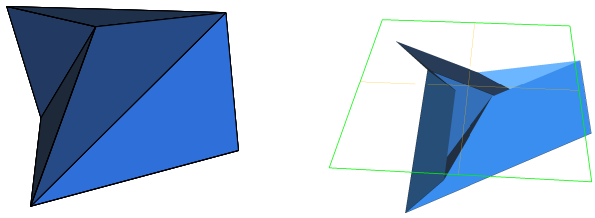


FIGURE 1 – polyédrique impossible de mailler sans point interne.

9.1.3 Les données pour construire un maillage

Pour construire un maillage, nous avons besoin de connaître :

- un ensemble de points

$$\mathcal{S} \stackrel{def}{=} \{x^i \in \mathbb{R}^2 / i \in \{1, ..., N_p\}\} \tag{13}$$

- un ensemble d'arêtes (couples de numéros de points) définissant le maillage de la frontière Γ_h des sous-domaines.

$$\mathcal{A} \stackrel{def}{=} \{(sa_1^j, sa_2^j) \in 1, ..., {N_p}^2 / j \in \{1, ..., N_a\}\} \tag{14}$$

- un ensemble de sous-domaines (composantes connexes de $\mathbb{R}^2 \backslash \Gamma_h$) à mailler, avec l'option par défaut suivante : mailler tous les sous-domaines bornés. Les sous-domaines peuvent être définis par une arête frontière et un sens (le sous domaine est à droite (-1) ou à gauche (+1) de l'arête orientée). Formellement, nous disposons donc de l'ensemble

$$\mathcal{SD} \stackrel{def}{=} \{(a^i, sens^i) \in \{1, ..., N_a\} \times \{-1, 1\} / i = 1, N_{sd}\} \tag{15}$$

  qui peut être vide (cas par défaut).

9.1.4 Triangulation et Convexifié

Construction d'un premier maillage, soit f une fonction strictement convexe de $\mathbb{R}^d$ à valeur de $\mathbb{R}$, soit G_f l'application graphe de f de $\mathbb{R}^d$ à valeur de $\mathbb{R}^{d+1}$ telle que $G_f : x = (x^1, .., x^d) \mapsto \tilde{x} = (x^1, .., x^d, f(x))$. et notons $e_{d+1} = (0, .., 0, 1) \in \mathbb{R}^{d+1}$.
Notons $\partial^- \mathcal{C}(G_f(S))$ la partie inférieure du bord du convexifié des points de S transformés par G_f, c'est-à-dire $\partial^- \mathcal{C}(G_f(S) = \{\tilde{x} \in \partial \mathcal{C}(G_f(S)) \ / \ n_{\tilde{x}}.e_{d+1} < 0\}$ où $n_{\tilde{x}}$ est la normal extérieur ,

Définition 9.4. *Le projeté $\partial^- \mathcal{C}(G_f(S))$ sur R^d défini une triangulation du convexifié de S que l'on notera $\mathcal{T}_f$.*

Donc, la construction de maillage est très liée à la construction de convexifie en dimension $d+1$. Je conjecture que quand f parcourt l'ensemble des fonction strictement convexe alors ; $\mathcal{T}_f$ parcourt l'ensemble des triangulations conformes de sommets S.

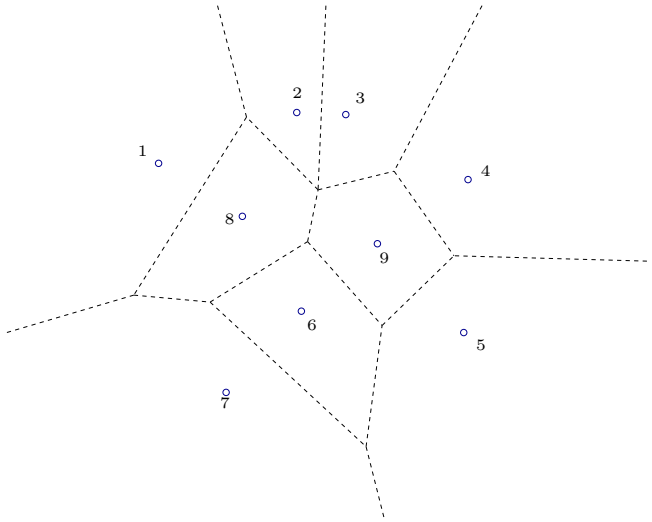


FIGURE 2 – Diagramme de Voronoï : les ensembles des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j.

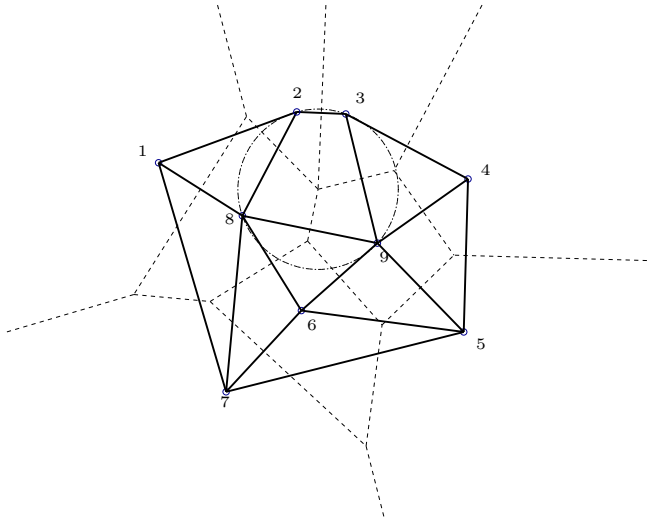


FIGURE 3 – Maillage de Delaunay : nous relions deux points x^i et x^j si les diagrammes V^i et V^j ont un segment en commun.

9.1.5 Maillage de Delaunay-Voronoï

La méthode est basée sur les diagrammes de Voronoï :

Définition 9.5. *Les diagrammes de Voronoï sont les polygones convexes $V^i, i = 1, N_p$ formés par l'ensemble des points de $\mathbb{R}^2$ plus proches de x^i que des autres points x^j (voir figure 2).*

On peut donc écrire formellement :

$$V^i \stackrel{def}{=} \{x \in \mathbb{R}^2 / ||x - x^i|| \leq ||x - x^j||, \forall j \in \{1, ..., N_p\}\}. \tag{16}$$

Ces polygones, qui sont des intersections finies de demi-espaces, sont convexes. De plus, les sommets v^k de ces polygones sont à égale distance des points $\{x^{i_j^k} / j = 1, ..., n_k\}$ de $\mathcal{S}$, où le nombre n_k est généralement égal (le cas standard) ou supérieur à 3. À chacun de ces sommets v^k, nous pouvons associer le polygone convexe construit avec les points $\{x^{i_j^k}, j = 1, ..., n_k\}$ en tournant dans le sens trigonométrique. Ce maillage est généralement formé de triangles, sauf si il y a des points cocycliques (voir figure 3 où $n_k > 3$).

Définition 9.6. *Nous appelons maillage de Delaunay strict, le maillage dual des diagrammes de Voronoï, construit en reliant deux points x^i et x^j, si les diagrammes V^i et V^j ont un segment en commun.*
Pour rendre le maillage triangulaire, il suffit de découper les polygones qui ne sont pas des triangles en triangles. Nous appelons ces maillages des maillages de Delaunay de l'ensemble $\mathcal{S}$.

La définition correct du maillage dual est quelque peut lourde. La voici, à chaque k-uplet de sommet $K = (x_{i_1}, ..., x_{i_k})$ on associe V_K l'intersection de cellules de Voronoï ($V_K = \cap_{j=1}^k V_{i_j}$) , si $dim(V_K) = d - k$ alors K est k-simplexe qui est une partie d'une maille de dimension k. L'on confondra K avec son convexifé. Pour définir complètement la maille, il faut traiter les cas des points co-sphérique, pour cela, on introduit sur les k-uplet tels que $dim(V_K) = d - k$. Les classes d'équivalences C_k pour la relation même V_K, c'est-à-dire $K \equiv K'$ si $V_K = V'_K$. Les éléments de dimension $d-k$ sont : les élément K_{c_k} quo sont l'union des éléments K de la classe c_k de C_k.

$$K_{c_k} = \bigcup_{K \in c_k} K.$$

Le démonstration de cette définition est laissé au lecteur.

Remarque 9. *Le domaine d'un maillage de Delaunay d'un ensemble de points $\mathcal{S}$ est l'intérieur du convexifié $\mathcal{C}(\mathcal{S})$ de l'ensemble de points $\mathcal{S}$.*

Nous avons le théorème suivant qui caractérise les maillages de Delaunay :

Théorème 9.2. *Un maillage $\mathcal{T}_{d,h}$ de Delaunay est tel que : pour tout triangle T du maillage, le disque ouvert $D(T)$ correspondant au cercle circonscrit à T ne contient aucun sommet (propriété de la boule vide). Soit, formellement :*

$$D(T) \cap \mathcal{T}_{0,h} = \emptyset. \tag{17}$$

Réciproquement, si le maillage $\mathcal{T}_{d,h}$ d'un domaine convexe vérifie la propriété de la boule vide, alors il est de Delaunay.

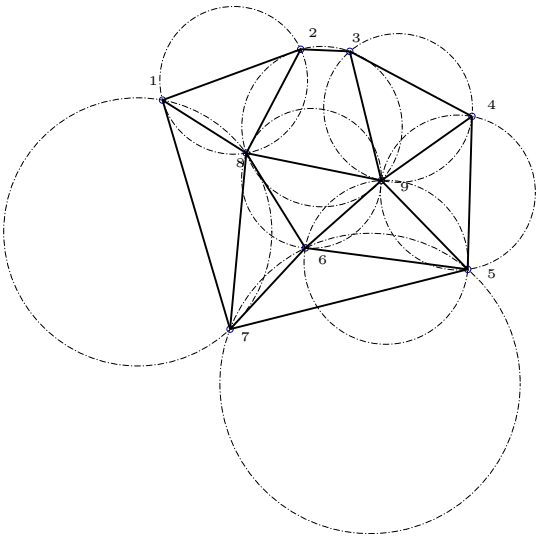


FIGURE 4 – Propriété de la boule vide : le maillage de Delaunay et les cercles circonscrits aux triangles.

Preuve : Montrons par l'absurde que si le maillage est de Delaunay alors il vérifie la propriété de la boule vide. Soit x^p un point de $\mathcal{S}$ dans la boule ouvert de d'élément $K = \{x^{k_1}, ...\}$ de centre c, on a $c \in V^{k_1}$ et $||c - x^p|| < ||c - x^{k_1}||$ ce qui est en contradiction avec la définition des V^k

Réciproquement : soit un maillage vérifiant (17).

Commençons par montrer la propriété (a) suivante : toutes les arêtes (x^i, x^j) du maillage sont telles que l'intersection $V^i \cap V^j$ contient les centres des cercles circonscrits aux triangles contenant $]x^i, x^j[$.

Soit une arête (x^i, x^j) ; cette arête appartient au moins à un triangle T. Notons c le centre du cercle circonscrit à T et montrons par l'absurde que c appartient à V^i et à V^j.

Si c n'est pas dans V^i, il existe un x^k tel que $||c - x^k|| < ||c - x^i||$ d'après (16), ce qui implique que x^k est dans le cercle circonscrit à T, d'où la contradiction avec l'hypothèse. Donc, c est dans V^i et il y en va de même pour V^j, ce qui démontre la propriété (a).

Il reste deux cas à étudier : l'arête est frontière ou est interne.

- si l'arête (x^i, x^j) est frontière, comme le domaine est convexe, il existe un point c' sur la médiatrice de x^i et x^j suffisamment loin du domaine dans l'intersection de V^i et V^j et tel que c' ne soit pas un centre de cercle circonscrit de triangle,
- si l'arête (x^i, x^j) est interne, elle est contenue dans un autre triangle T' et $V^i \cap V^j$ contient aussi c', le centre du cercle circonscrit à T'.

Dans tous les cas, c et c' sont dans $V^i \cap V^j$ et comme V^i et V^j sont convexes, l'intersection $V^i \cap V^j$ est aussi convexe. Donc le segment $[c, c']$ est inclus dans $V^i \cap V^j$.

Maintenant, il faut étudier les deux cas : $c = c'$ ou $c \neq c'$.

- si le segment $[c, c']$ n'est pas réduit à un point, alors l'arête (x^i, x^j) est dans le maillage Delaunay ;
- si le segment $[c, c']$ est réduit à un point, alors nous sommes dans cas où l'arête (x^i, x^j) est interne et c' est le centre de $D(T')$. Les deux triangles T, T' contenant l'arête (x^i, x^j) sont cocycliques et l'arête n'existe pas dans le maillage de Delaunay strict.

Pour finir, les arêtes qui ne sont pas dans le maillage de Delaunay sont entre des triangles

cocycliques. Il suffit de remarquer que les classes d'équivalence des triangles cocycliques d'un même cercle forment un maillage triangulaire de polygones du maillage de Delaunay strict. ∎

Cette démonstration est encore valide en dimension d quelconque, en remplaçant les arêtes par des hyperfaces qui sont de codimension 1.

Remarque 10. ‖ *Il est possible d'obtenir un maillage de Delaunay strict en changeant la propriété de la boule vide définie en (17) par la propriété stricte de la boule vide, définie comme suit :*

$$\overline{D(T)} \cap \mathcal{T}_{0,h} = \overline{T} \cap \mathcal{T}_{0,h}, \tag{18}$$

où $\overline{D(T)}$ est le disque fermé correspondant au cercle circonscrit à un triangle T et $\mathcal{T}_{0,h}$ est l'ensemble de sommets du maillage. La différence entre les deux propriétés (18) et (17) est qu'il peut exister dans (17) d'autres points de $\mathcal{T}_{0,h}$ sur le cercle circonscrit $C(T) = \overline{D(T)} \setminus D(T)$.

B. Delaunay a montré que l'on pouvait réduire cette propriété au seul motif formé par deux triangles adjacents.

Lemme 9.7 (Delaunay)**.** *Si le maillage $\mathcal{T}_{d,h}$ d'un domaine convexe est tel que tout sous-maillage formé de deux triangles adjacents par une arête vérifie la propriété de la boule vide, alors le maillage $\mathcal{T}_{d,h}$ vérifie la propriété globale de la boule vide et il est de Delaunay.*

La démonstration de ce lemme est basée sur

Alternative 1. *Si deux cercles C_1 et C_2 s'intersectent sur la droite D séparant le plan des deux demi-plans P^+ et P^-, alors on a l'alternative suivante :*

$$D_1 \cap P^+ \subset D_2 \text{ et } D_2 \cap P^- \subset D_1,$$

ou

$$D_2 \cap P^+ \subset D_1 \text{ et } D_1 \cap P^- \subset D_2,$$

où D_i est le disque associé au cercle C_i, pour $i = 1, 2$.

Exercice 13. ‖ La démonstration de l'alternative est laissée en exercice au lecteur.

Preuve du lemme de Delaunay: nous faisons une démonstration par l'absurde qui est quelque peu technique.
Supposons que le maillage ne vérifie pas la propriété de la boule vide. Alors il existe un triangle $T \in \mathcal{T}_{d,h}$ et un point $x^i \in \mathcal{T}_{0,h}$, tel que $x^i \in D(T)$.
Soit a un point interne au triangle T tel que $]a, x^i[$ ne soit tangent à aucun élément T' de $\mathcal{T}_{d,h}$ (il suffit de déplacer un petit peu le point a si ce n'est pas le cas). Le segment $]a, x^i[$ est inclus dans le domaine car il est convexe. Nous allons lui associer l'ensemble des triangles $T_a^j, j = 0, .., k_a$ qui intersectent le segment$]a, x^i[$. Cette ensemble est une chaîne de triangles T_a^j pour $j = 0, .., k_a$ c'est à dire que les triangles T_a^j et T_a^{j+1} sont adjacents par une arête en raison de l'hypothèse de non tangence.
Nous pouvons toujours choisir le couple (T, x^i) tel que $x^i \in D(T)$, $x^i \notin T$ et tel que le cardinal k de la chaîne soit minimal. Le lemme se résume à montrer que $k = 1$.
Soit x^{i_1} (resp. x^{i_0}) le sommet de T^1 (resp. T^0) opposé à l'arête $T^0 \cap T^1$.

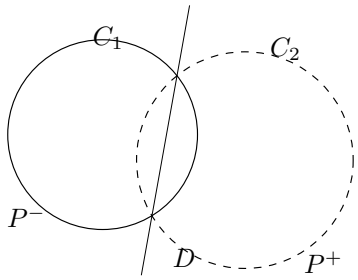


FIGURE 5 – Représentation graphique de l'alternative 1.

— Si x^{i_0} est dans $D(T^1)$ alors $k = 1$ (la chaîne est T^1, T^0).
— Si x^{i_1} est dans $D(T^0)$ alors $k = 1$ (la chaîne est T^0, T^1)
— Sinon, les deux points x^{i_0} et x^{i_1} sont de part et d'autre de la droite D définie par l'intersection des deux triangles T^0 et T^1, qui est aussi la droite d'intersection des deux cercles $C(T^0)$ et $C(T^1)$. Cette droite D définit deux demi-plans $\mathcal{P}^0$ et $\mathcal{P}^1$ qui contiennent, respectivement, les points x^{i_0} et x^{i_1}. Pour finir, il suffit d'utiliser l'alternative 1 avec les cercles $C(T^0)$ et $C(T^1)$. Comme x^{i_0} n'est dans $D(T^1)$, alors $D(T^0)$ est inclus dans $D(T^1) \cap \mathcal{P}^0$. Mais, $x^i \in C(T^0)$ par hypothèse, et comme x^i n'est pas dans le demi-plan $\mathcal{P}^1$ car le segment $]a, x^i[$ coupe la droite D, on a $x^i \in C(T^0) \subset C(T^1)$, ce qui impliquerait que le cardinal de la nouvelle chaîne est $k-1$ et non k d'où la contradiction avec l'hypothèse de k minimal.

∎

Remarque 11. *La démonstration du lemme de Delaunay est encore valide en dimension n ; il suffit de remplacer cercle par sphère, droite par hyperplan et arête par hyperface.*
Mais, attention, en dimension 3, il existe des configurations formées de deux tétraèdres adjacentes par une face non-convexe ne vérifiant pas la propriété de la boule vide .

Nous en déduisons une autre caractérisation du maillage de Delaunay :

Théorème 9.3. *La triangulation $\mathcal{T}_f$ de la définition 9.4 est la triangulation de Delaunay si $f(x) = ||x||^2$ et les maillages de Delaunay sont les uniques triangulations rendant convexe l'interpolé P_1 Lagrange de la fonction $x \to ||x||^2$.*

Preuve : Il suffit de montre le théoreme pour tous le maillages formés de deux éléments adjacent T_a et T_b par une hyper-face F de sommet $(p^1, .., p^d)$ et de sommets opposé a (resp. b), grâce au lemme de Delaunay. Nos deux éléments T_a et T_b(d-simplex positif) s'écrivent $T_a = (p^1, .., p^d, a)$ et $T_b = (p^2, p^1, p^3, .., p^d, b)$, notons p le point l'intersection $p = A(p^1, .., p^d) \cap [a, b]$ où $A(p^1, .., p^d)$ est l'espace affine engendre par les points $p^1, .., p^d$ et notons λ la coordonnée barycentrique telle que $p = \lambda a + (1 - \lambda)b$ et les coordonnées barycentriques $\mu_1, .., \mu_d$ tel que $p = \sum \mu_i p^i$ et tel que $\sum \mu_i = 1$.
L'interpolé est convexe si seulement si on a

$$\sum \mu_i ||p^i||^2 \leq ||a||^2 \lambda + ||b||^2 (1 - \lambda). \tag{19}$$

• Pour application affine g de $\mathbb{R}^d$ dans R, l'ensemble des zéros de $||x||^2 - g(x)$ est soit le $\emptyset$, un point, ou une sphère, et $||x||^2 - g(x) \leq 0$ si et seulement x est dans la sphère. (à démontrer en exercice).
•En appliquant le point précédent avec g égal à l'équation de affine telle $g(q_i) = ||q^i||^2$ pour $i = 0, ..., d$ (d'où $||q^i||^2 - g(q^i) = 0$). On a donc (19) si et seulement si b est dans la sphère inscrit de T_a, ce qui acheve la démonstration. Nous venons de dire que le $(d+1)$ simplex $(G_f(p^1), ..., G_f(p^d), G_f(a), G_f(b))$ est positif.

∎

Nous avons retrouver la formule [George,Borouchaki-1997, equation (1.13)] qui décrit la boule circonscrite au d-simplex positif $(q^0, \ldots, q^d)$. le point b est dans la boule si et seulement si

$$det \begin{vmatrix} q_1^0 & \ldots & q_1^d & b_1 \\ \vdots & \ldots & \vdots & \vdots \\ q_d^0 & \ldots & q_d^d & b_d \\ ||q^0||^2 & \ldots & ||q^0||^2 & ||b||^2 \\ 1 & \ldots & 1 & 1 \end{vmatrix} \geq 0 \qquad (20)$$

En fait, nous avons montré que nous pouvons réduire cette propriété au seul motif formé par deux triangles adjacents. De plus, comme un quadrilatère non-convexe maillé en deux triangles vérifie la propriété de la boule vide, il suffit que cette propriété soit vérifiée pour toutes les paires de triangles adjacents formant un quadrilatère convexe.

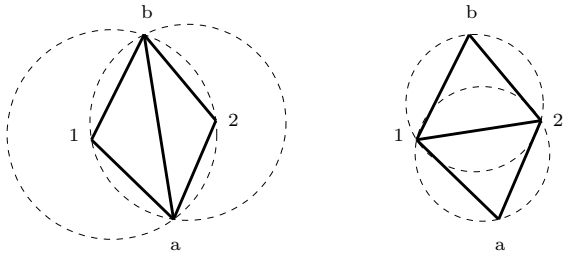


FIGURE 6 – Échange de diagonale d'un quadrilatère convexe selon le critère de la boule vide.

Nous ferons un échange de diagonale $[s^a, s^b]$ dans un quadrilatère convexe de coordonnées s^1, s^a, s^2, s^b (tournant dans le sens trigonométrique) si le critère de la boule vide n'est pas vérifié comme dans la figure 6.

Lemme 9.8. *Le critère de la boule vide dans un quadrilatère convexe s^1, s^a, s^2, s^b en $[s^1, s^2]$ est équivalent à l'inégalité angulaire (propriété des angles inscrits dans un cercle) :*

$$\widehat{s^1 s^a s^b} < \widehat{s^1 s^2 s^b}.$$

Preuve : comme la cotangente est une fonction strictement décroissante entre $]0, \pi[$, il suffit de vérifier :

$$cotg(\widehat{s^1 s^a s^b}) = \frac{(s^1 - s^a, s^b - s^a)}{det(s^1 - s^a, s^b - s^a)} \quad > \quad \frac{(s^1 - s^2, s^b - s^2)}{det(s^1 - s^2, s^b - s^2)} = cotg(\widehat{s^1 s^2 s^b}),$$

où $(.,.)$ est le produit scalaire de $\mathbb{R}^2$ et $det(.,.)$ est le déterminant de la matrice formée avec les deux vecteurs de $\mathbb{R}^2$.

Ou encore, si l'on veut supprimer les divisions, on peut utiliser les aires des triangles $aire^{1ab}$ et $aire^{12b}$. Comme

$$det(s^1 - s^a, s^b - s^a) = 2 \times aire^{1ab} \quad \text{et} \quad det(s^1 - s^2, s^b - s^2) = 2 \times aire^{12b},$$

le critère d'échange de diagonale optimisé est

$$aire^{12b} \quad (s^1 - s^a, s^b - s^a) \quad > \quad aire^{1ab} \quad (s^1 - s^2, s^b - s^2). \tag{21}$$

∎

Maintenant, nous avons théoriquement les moyens de construire un maillage passant par les points donnés, mais généralement nous ne disposons que des points de la frontière ; il va falloir donc générer ultérieurement les points internes du maillage.

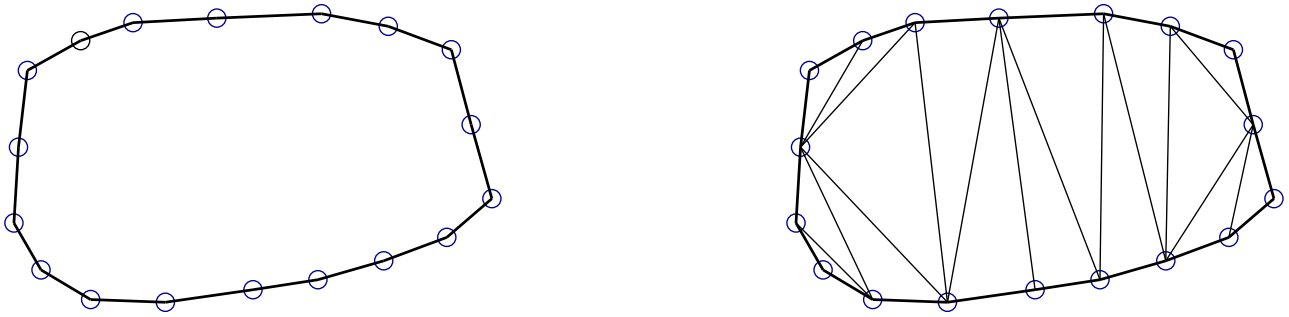


FIGURE 7 – Exemple de maillage d'un polygone sans point interne.

Par construction, le maillage de Delaunay n'impose rien sur les arêtes. Il peut donc arriver que ce maillage ne respecte pas la discrétisation de la frontière, comme nous pouvons le remarquer sur la figure 8. Pour éviter ce type de maillage, nous pouvons suivre deux pistes :

- modifier le maillage afin qu'il respecte la frontière ;
- ou bien, modifier la discrétisation de la frontière afin qu'elle soit contenue dans le maillage de Delaunay.

Pour des raisons de compatibilité avec d'autres méthodes nous ne modifierons pas le maillage de la frontière.

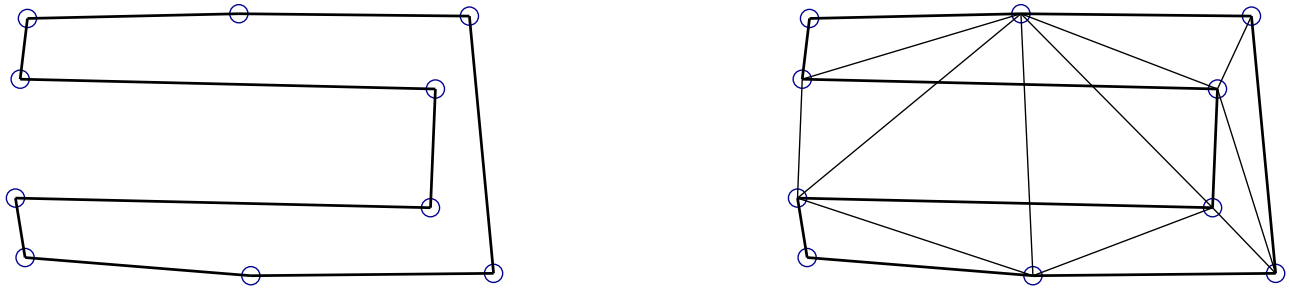


FIGURE 8 – Exemple de maillage de Delaunay ne respectant pas la frontière.

9.1.6 Forçage de la frontière

Comme nous l'avons déjà vu, les arêtes de la frontière ne sont pas toujours dans le maillage de Delaunay construit à partir des points frontières (voir figures 8 et 9).
Nous dirons qu'une arête (a, b) coupe une autre arête (a', b') si $]a, b[\cap]a', b'[= p \neq \emptyset$.

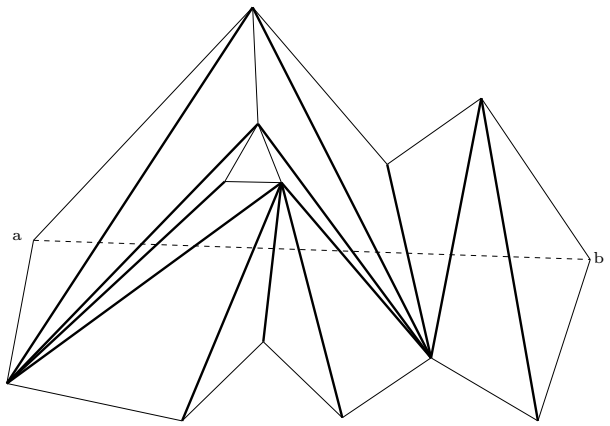


FIGURE 9 – Exemple d'arête $]a,b[$ manquante dans un maillage compliqué.

Théorème 9.4. *Soient $\mathcal{T}_{d,h}$ une triangulation et a,b deux sommets différents de $\mathcal{T}_{d,h}$ (donc dans $\mathcal{T}_{0,h}$) tels que*

$$]a,b[\cap\mathcal{T}_{0,h}=\emptyset \quad \text{et} \quad]a,b[\subset\mathcal{O}_h. \tag{22}$$

Alors, il existe une suite finie d'échanges de diagonale de quadrilatère convexe, qui permet d'obtenir un nouveau maillage $\mathcal{T}_{d,h}^{ab}$ contenant l'arête (a,b).
Nous avons de plus la propriété de localité optimale suivante : toute arête du maillage $\mathcal{T}_{d,h}$ ne coupant pas $]a,b[$ est encore une arête du nouveau maillage $\mathcal{T}_{d,h}^{ab}$.

Preuve : nous allons faire un démonstration par récurrence sur le nombre $m_{ab}(\mathcal{T}_{d,h})$ d'arêtes du maillage $\mathcal{T}_{d,h}$ coupant l'arête (a,b).
Soit T^i, pour $i=0,...,m_{ab}(\mathcal{T}_{d,h})$, la liste des triangles coupant $]a,b[$ tel que les traces des T^i sur $]a,b[$ aillent de a à b pour $i=0,...,n$.
Comme $]a,b[\cap\mathcal{T}_{0,h}=\emptyset$, l'intersection de $\overline{T}^{i-1}$ et $\overline{T}^{i}$ est une arête notée $[\alpha_i,\beta_j]$ qui vérifie

$$[\alpha_i,\beta_j] \overset{def}{=} \overline{T}^{i-1}\cap\overline{T}^{i}, \quad \text{avec} \quad \alpha_i\in P_{ab}^{+} \quad \text{et} \quad \beta_i\in P_{ab}^{-}, \tag{23}$$

où P_{ab}^{+} et P_{ab}^{-} sont les deux demi-plans ouverts, définis par la droite passant par a et b.
Nous nous placerons dans le maillage restreint $\mathcal{T}_{d,h}^{r_{a,b}}$ formé seulement de triangles T^i pour $i=0,...,m_{ab}(\mathcal{T}_{d,h})=m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})$ pour assurer le propriété de localité. De plus, le nombre de triangles N_t^{ab} de $\mathcal{T}_{d,h}^{a,b}$ est égal à $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})+1$.

- Si $m_{ab}(\mathcal{T}_{d,h}^{r_{ab}})=1$, on fait l'observation que le quadrilatère formé par les deux triangles contenant l'unique arête coupant (a,b) est convexe, et donc il suffit d'échanger les arêtes du quadrilatère.
- Sinon, supposons vraie la propriété pour toutes les arêtes (a,b) vérifiant (22) et telles que $m_{ab}(\mathcal{T}_{d,h}^{r_{ab}})<n$ et ceci pour tous les maillages possibles.

Soit une arête (a,b) vérifiant (22) et telle que $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b}})=n$. Soit α_{i^+} le sommet α_i pour $i=1,...,n$ le plus proche du segment $[a,b]$. Nous remarquerons que les deux inclusions suivantes sont satisfaites :

$$]a,\alpha_{i^+}[\subset \overset{\circ}{\overbrace{\bigcup_{i=0}^{i^+-1}\overline{T}^{i}}} \quad \text{et} \quad]\alpha_{i^+},b[\subset \overset{\circ}{\overbrace{\bigcup_{i=i^+}^{n}\overline{T}^{i}}}. \tag{24}$$

Les deux arêtes $]a, \alpha_{i^+}[$ et $]\alpha_{i^+}, b[$ vérifient les hypothèses de récurrence, donc nous pouvons les forcer par échange de diagonales, car elles ont des supports disjoints. Nommons $\mathcal{T}_{d,h}^{r_{a,b^+}}$ le maillage obtenu après forçage de ces deux arêtes. Le nombre de triangles de $\mathcal{T}_{d,h}^{r_{a,b^+}}$ est égal à $n+1$ et, par conséquent, $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b^+}}) \leq n$.
Il nous reste à analyser les cas suivants :

- si $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b^+}}) < n$, nous appliquons l'hypothèse de récurrence et la démonstration est finie ;
- si $m_{ab}(\mathcal{T}_{d,h}^{r_{a,b^+}}) = n$, nous allons forcer (a, b) dans le maillage $\mathcal{T}_{d,h}^{r_{a,b^+}}$ et utiliser la même méthode. Les T^i seront maintenant les triangles de $\mathcal{T}_{d,h}^{r_{a,b^+}}$ et les α^+ en β^+ seront définis par (23). Nous avons traité le demi-plan supérieur, traitons maintenant la partie inférieure. Soit i^- l'indice tel que le sommet $\beta_{i^+}^+$ soit le plus proche du segment $]a, b[$ des sommets β_i^+. Nous forçons les deux arêtes $]a, \beta_{i^-}[$ et $]\beta_{i^-}, b[$ en utilisant les mêmes arguments que précédemment.

Nous avons donc un maillage local du quadrilatère $a, \alpha_{i^+}, b, \beta_{i^-}^+$ qui contient l'arête $]a, b[$ et qui ne contient aucun autre point du maillage. Il est donc formé de deux triangles T, T' tels que $]a, b[\subset \overline{T} \cup \overline{T}'$, ce qui nous permet d'utiliser une dernière fois l'hypothèse de récurrence ($n = 1$) pour finir la démonstration. ∎

Remarque 12. *On en déduit facilement une autre démonstration du théorème 9.1 : il suffit de prendre un maillage de Delaunay de l'ensemble des sommets de l'ouvert, de forcer tous les segments frontières de l'ouvert et de retirer les triangles qui ne sont pas dans l'ouvert.*

Du théorème 9.4, il découle :

Théorème 9.5. *Soit deux maillages $\mathcal{T}_{d,h}$ et $\mathcal{T}'_{d,h}$ ayant les mêmes sommets ($\mathcal{T}_{0,h} = \mathcal{T}'_{0,h}$) et le même maillage du bord $\partial\mathcal{T}_{d,h} = \partial\mathcal{T}'_{d,h}$. Alors, il existe une suite d'échanges de diagonales de quadrilatères convexes qui permet de passer du maillage $\mathcal{T}_{d,h}$ au maillage $\mathcal{T}'_{d,h}$.*

Preuve : il suffit de forcer toutes les arêtes du maillage $\mathcal{T}_{d,h}$ dans $\mathcal{T}'_{d,h}$. ∎

Pour finir cette section, donnons un algorithme de forçage d'arête très simple (il est dû à Borouchaki [George,Borouchaki-1997, page 99]).

Algorithme 7.

Forçage d'une arête dans un maillage

Si l'arête (s^a, s^b) n'est pas une arête du maillage de Delaunay, nous retournons les diagonales (s^α, s^β) des quadrangles convexes $s^\alpha, s^1, s^\beta, s^2$ formés de deux triangles dont la diagonale $]s^\alpha, s^\beta[$ coupe $]s^a, s^b[$ en utilisant les critères suivants :

- *si l'arête $]s^1, s^2[$ ne coupe pas $]s^a, s^b[$, alors on fait l'échange de diagonale ;*
- *si l'arête $]s^1, s^2[$ coupe $]s^a, s^b[$, on fait l'échange de diagonale de manière aléatoire.*

Comme il existe une solution au problème, le fait de faire des échanges de diagonales de manière aléatoire va permettre de converger, car, statistiquement, tous les maillages possibles sont parcourus car ils sont en nombre fini.

9.1.7 Recherche de sous-domaines

L'idée est de repérer les parties qui sont les composantes connexes de $\mathcal{O}_h \setminus \cup_{j=1}^{N_a}[x^{sa_1^j}, x^{sa_2^j}]$, ce qui revient à définir les composantes connexes du graphe des triangles adjacents où l'on a supprimé les connexions avec les arêtes de $\mathcal{A}$. Pour cela, nous utilisons l'algorithme de coloriage qui recherche la fermeture transitive d'un graphe.

Algorithme 8. **Coloriage de sous-domaines**

```
Coloriage(T)
 Si  T n'est pas colorié
   Pour tous les triangles  T' adjacents à  T par une arête non-marquée
    Coloriage(T')
marquer toutes les arêtes T_{d,h} qui sont dans  A
Pour tous les Triangles  T non-coloriés
 Changer de couleur
  Coloriage(T)
```

Observons qu'à chaque couleur correspond une composante connexe de $\mathcal{O}_h \setminus \cup_{j=1}^{N_a}[x^{sa_1^j}, x^{sa_2^j}]$. La complexité de l'algorithme est en $3 \times N_t$, où N_t est le nombre de triangles.
Attention, si l'on utilise simplement la récursivité du langage, nous risquons de gros problèmes car la profondeur de la pile de stockage mémoire est généralement de l'ordre du nombre d'éléments du maillage.

Exercice 14. Récrire l'algorithme 8 sans utiliser la récursivité.

9.1.8 Génération de points internes

Pour compléter l'algorithme de construction du maillage, il faut savoir générer les points internes (voir figure 4). Le critère le plus naturel pour distribuer les points internes est d'imposer en tout point $\mathbf{x}$ de $\mathbb{R}^2$, le pas de maillage $h(\mathbf{x})$.
La difficulté est que, généralement, cette information manque et il faut construire la fonction $h(\mathbf{x})$ à partir des données du maillage de la frontière. Pour ce faire, nous pouvons utiliser, par exemple, la méthode suivante :

1. À chaque sommet s de $\mathcal{T}_{0,h}$ on associe une taille de maillage qui est la moyenne des longueurs des arêtes ayant comme sommet s. Si un sommet de $\mathcal{T}_{0,h}$ n'est contenu dans aucune arête, alors on lui affecte une valeur par défaut, par exemple le pas de maillage moyen.
2. On construit le maillage de Delaunay de l'ensemble des points.
3. Pour finir, on calcule l'interpolé P^1 (voir §**??**, définition 11) de la fonction $h(\mathbf{x})$ dans tous les triangles de Delaunay.

Dans la pratique, nous préférons disposer d'une fonction $N(a,b)$ de $\mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ qui donne le nombre de mailles entre les points a et b. Nous pouvons construire différents types de fonctions N à partir de $h(\mathbf{x})$. Par exemple, une expression utilisant la moyenne simple $|ab|^{-1}\int_a^b h(t)dt$ serait :

$$N_1(a,b) \quad \stackrel{def}{=} \quad |ab|(\frac{\int_a^b h(t)dt}{|ab|})^{-1} = (\int_a^b h(t)dt)^{-1}. \tag{25}$$

Nous pouvons aussi utiliser quelques résultats de la géométrie différentielle. La longueur l d'une courbe γ paramétrée par $t \in [0,1]$ étant définie par

$$l \stackrel{def}{=} \int_{t=0}^{1} \sqrt{(\gamma'(t), \gamma'(t))} dt, \tag{26}$$

si nous voulons que la longueur dans le plan tangent en $\mathbf{x}$ d'un segment donne le nombre de pas, il suffit d'introduire la longueur *remmanienne* suivante (on divise par $h(\mathbf{x})$) :

$$l^h \stackrel{def}{=} \int_{t=0}^{1} \frac{\sqrt{(\gamma'(t), \gamma'(t))}}{h(\gamma(t))} dt \tag{27}$$

Nous obtenons une autre définition de la fonction $N(a,b)$:

$$N_2(a,b) \stackrel{def}{=} \int_a^b h(t)^{-1} dt. \tag{28}$$

Les deux méthodes de construction de $N(a,b)$ sont équivalentes dans le cas où $h(\mathbf{x})$ est indépendant de $\mathbf{x}$. Regardons ce qui change dans le cas d'une fonction h affine sur le segment $[0,l]$. Si

$$h(t) = h_0 + (h_l - h_0)t, \quad t \in [0,1],$$

nous obtenons

$$N_1(0,t) = \left(\int_0^t [h_0 + (h_l - h_0)x] dx \right)^{-1} = (h_0 t + \frac{(h_l - h_0)}{2} t^2)^{-1}, \tag{29}$$

$$N_2(0,t) = \frac{1}{h_l - h_0} \ln \left(1 + \frac{h_l - h_0}{h_0} t \right). \tag{30}$$

Par construction, N_2 vérifie la relation de Chasle :

$$N_2(a,c) = N_2(a,b) + N_2(b,c),$$

où b est un point entre a et b, alors que N_1 ne vérifie clairement pas cette relation. C'est la raison pour laquelle nous préférons l'expression (30) pour la fonction N. De plus, si nous voulons avoir un nombre optimal de points tels que $N_2(0,t) = i, \forall i \in 1, 2, ..$ dans (30), ces points seront alors distribués suivant une suite géométrique de raison $\frac{\ln(h_l - h_0)}{h_l - h_0}$.

9.2 Algorithme de construction du maillage

Pour construire un maillage de Delaunay à partir d'un ensemble donné de points, nous allons suivre les étapes suivantes :

1. Trier les $\mathbf{x}^i$ en ordre par rapport à la norme $||\mathbf{x}^i||$. Cet ordre est tel que le point courant ne soit jamais dans le convexifié des points précédents.
2. Ajouter les points un à un suivant l'ordre prédéfini à l'étape 2. Les points ajoutés sont toujours à l'extérieur du maillage courant. Donc, on peut créer un nouveau maillage en reliant toutes les arêtes frontières du maillage précédent qui voit les points du bon côté.

3. Pour que le maillage soit de Delaunay, il suffit d'appliquer la méthode **d'optimisation locale** du maillage suivant : autour d'un sommet rendre de Delaunay tous les motifs formés de deux triangles convexes contenant ce sommet, en utilisant la formule (21). Puis, appliquer cette optimisation à chaque nouveau point ajouté pour que le maillage soit toujours de Delaunay.
4. Forcer la frontière dans le maillage de Delaunay en utilisant l'algorithme 7.
5. Retirer les parties externes du domaine en utilisant l'algorithme 8.
6. Générer des points internes. S'il y a des points internes nous recommençons avec l'étape 1, avec l'ensemble de points auquel nous avons ajouté les points internes.
7. Régularisation, optimisation du maillage.

9.3 Recherche d'un triangle contenant un point

fonction qui recherche un triangle K de $\mathcal{T}_{d,h}$ contenant un point $B = (x, y)$ à partir d'un triangle de départ T défini. L'algorithme utilisé est le suivant :

Algorithme 9.

Recherche d'un triangle contenant un point

Partant de du triangle K_s=T, pour les trois arêtes $(a_i, b_i), i = 0, 1, 2$ du triangle K, tournant dans le sens trigonométrique.
Calculer l'aire des trois triangles (a_i, b_i, p)
- *si les trois aires sont positives alors $p \in K$ (stop),*
- *sinon nous choisirons comme nouveau triangle K l'un des triangles adjacent à l'une des arêtes associées à une aire négative (les ambiguïtés sont levées aléatoirement).*

Si le sommet recherché est à l'extérieur, nous retournerons une arête frontière (ici un triangle dégénéré).

Exercice 15. ‖ Prouver que l'algorithme 9 précédent marche si le maillage est convexe.

Il existe des cas où les triangles ne sont pas connue explicitement, nous disposons que du graphe de la triangulation, c'est à dire que par sommet, on a la listes circulaire des sommets voisins tournant dans le sens trigonométrique.
Voilà une version

Algorithme 10.

Recherche d'un triangle contenant un point, version sans triangle

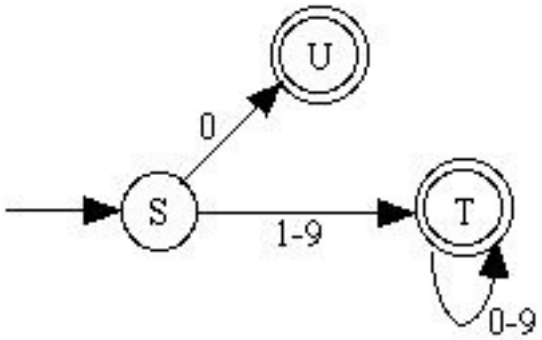
Partant d'un sommet s, trouver les 2 arêtes concécutive (s, a) et (s, b) du sommet s (où a et b sont les 2 autres sommets des arêtes), tournant dans le sens trigonométrique, tel que les aires signées des 2 triangles $A = aire(a, p, s)$ et $B = aire(b, s, p)$ soit positive.
Soit l'aire du triangle $C = aire(a, b, s)$ si C est négatif ou nul alors le point est à l'exterieur, si $C - A - B$ est positif alors on a trouve le triangle sinon choisir aléatoirement le nouveau point s entre les points a et b et continuer.

Exercice 16. ‖ Prouver que l'algorithme 10 précédent marche si le maillage est convexe.

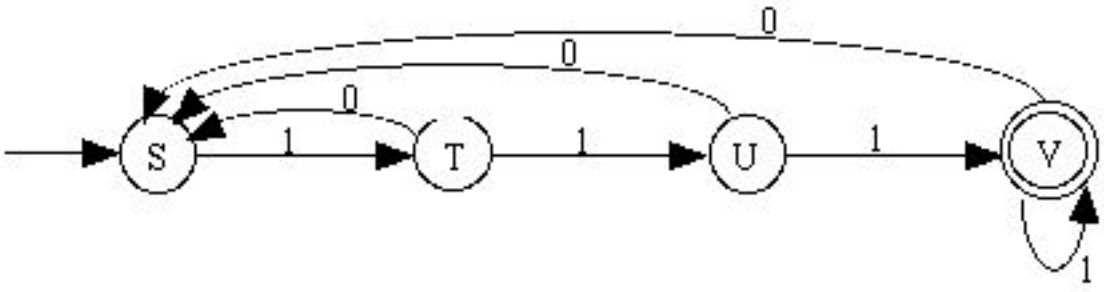
10 Automates finis

@ est un quintuplet $(\mathcal{S}, \mathcal{A}, S, \tau, f)$, où :
- $\mathcal{S}$ est un ensemble fini dont les éléments sont appelés des états,
- $\mathcal{A}$ est un alphabet,
- S est un état particulier, appelé état initial,
- τ est une fonction de $\mathcal{S} \times \mathcal{A}$ dans $\mathcal{S}$, appelée fonction de transition,
- f est une fonction de $\mathcal{S}$ à valeurs dans l'ensemble à deux éléments {succès, échec} (on dira qu'un état X est acceptant si $f(X)$ est succès).

Il faut comprendre l'automate comme une machine qui permet d'attribuer une valeur, succès ou échec, à chaque mot α de $\mathcal{A}^*$. Au départ, la machine est, comme on peut s'en douter, dans l'état initial. On lit de gauche à droite les caractères de α, et chaque caractère fait passer la machine d'un état à un autre selon la fonction de transition. Quand le mot a été entièrement lu, la machine se trouve dans un état X, et on dira que la machine *accepte* ou *reconnaît* α si X est acceptant.
Le langage *associé à* ou *reconnu par* un automate est l'ensemble des mots acceptés par l'automate.
On appelle *état d'erreur* un état non acceptant E tel que, quel que soit a dans $\mathcal{A}$, $\tau(a, E)$ soit égal à E. En d'autres termes, si au cours de la lecture d'un mot α, on entre dans un état d'erreur, il est inutile de poursuivre la lecture car α ne sera pas accepté.
Pour représenter les automates finis, on utilise généralement un graphe plan orienté dont les sommets correspondent aux états et les flèches sont étiquetées par les caractères correspondant aux transitions. Les états acceptants sont entourés deux fois, l'état de départ est signalé par une flèche entrante. Finalement, pour alléger le dessin, il est entendu de ne pas représenter les états d'erreur ainsi que les transitions qui y mènent.



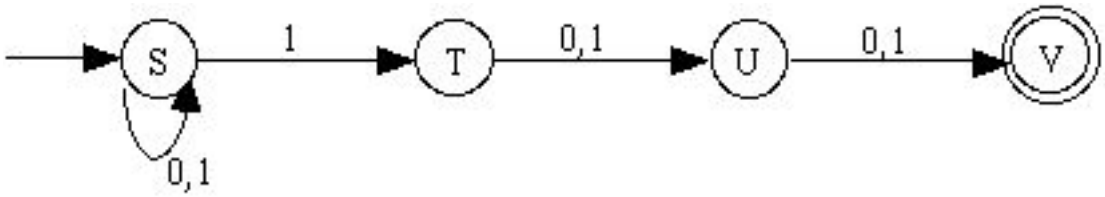
Exemple 1*:*
Le langage reconnu par cet automate est le même que celui engendré par la grammaire de l'Exemple 3.1.2, c'est-à-dire les nombres entiers positifs en écriture décimale.



Exemple 2*:*
Le langage reconnu par cet automate est l'ensemble des chaînes de 0 et 1 dont les trois derniers caractères sont des 1.

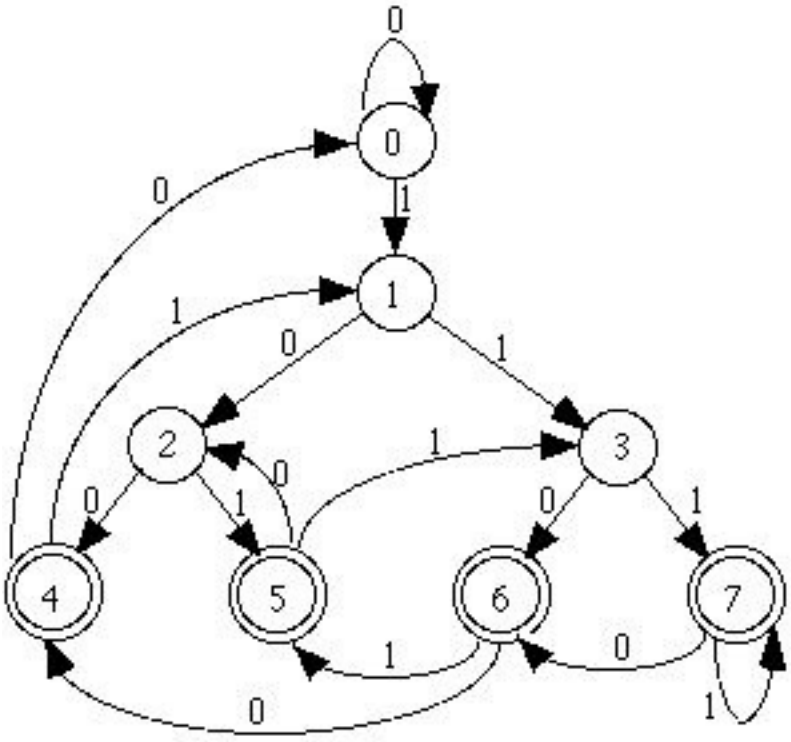
On obtient la notion d'automate (non déterministe) fini en autorisant l'existence de plusieurs flèches partant du même état et portant la même étiquette, ainsi que de flèches sans étiquettes ou plutôt étiquetées ε, que l'on désignera sous le nom d'ε-transitions.
Un chemin dans un tel automate est une suite de flèches consécutives, et le mot associé à ce chemin est formé par les étiquettes des flèches. Le langage reconnu par l'automate est l'ensemble des mots associés aux chemins menant de l'état initial à un état acceptant.
On observe que la terminologie est légèrement incorrecte, bien que traditionnelle, puisqu'un automate déterministe n'est qu'un cas particulier d'automate non déterministe. Contrairement à ce que l'on pourrait penser, l'ensemble des langages reconnus par des automates non déterministes n'est pas plus grand que celui des langages reconnus par des automates déterministes. En effet, la procédure suivante associe à tout automate fini $(\mathcal{S}, \mathcal{A}, S, \tau, f)$ un autre automate $(\mathcal{S}', \mathcal{A}', S', \tau', f')$, déterministe, qui reconnaît le même langage :
- *si $\mathcal{S}$ est l'ensemble des états du premier automate, on prendra $\mathcal{S}'$ égal à l'ensemble $\mathcal{P}(\mathcal{S})$ des parties de $\mathcal{S}$; on choisit aussi $\mathcal{A}'$ égal à l'alphabet $\mathcal{A}$;*
- *si P est un élément de $\mathcal{S}'$ et a appartient à l'alphabet $\mathcal{A}$, $\tau'(a, P)$ est l'ensemble des extrémités des chemins du premier automate qui sont associés au mot a et dont l'origine est dans P (c'est-à-dire l'ensemble des états où l'on peut arriver en partant d'un état de P et en parcourant une suite finie de flèches dont une et une seule est étiquetée a et les autres sont étiquetées ε) ;*
- *l'état initial S' est l'ensemble des états où l'on peut aboutir à partir de l'état initial S du premier automate par une suite finie d'ε-transitions, en particulier il contient S ;*
- *enfin, un élément P de $\mathcal{S}'$ est acceptant s'il contient un état acceptant du premier automate.*

On constate en effet que, dans le nouvel automate $(\mathcal{S}', \mathcal{A}', S', \tau', f')$, l'état dans lequel on se trouve après lecture du mot α est bien l'ensemble des états possibles après lecture de α dans le premier automate ; α est donc reconnu par un des deux automates si et seulement si il est reconnu par l'autre.



Exemple 3*:*
Cet automate est non déterministe puisque deux flèches étiquetées 1 partent de S. Le langage qu'il reconnaî t est formé des chaînes de $\mathcal{A} = \{0, 1\}$ comportant au moins trois caractères et dont le troisième caractère à partir de la fin est un 1. En appliquant l'algorithme précédent, on constate que seules les 8 parties de $\{S, T, U, V\}$ contenant S interviennent. Dans le schéma ci-après, elles sont numérotées de la façon suivante :

$$0 : \{S\}, \quad 1 : \{S, T\}, \quad 2 : \{S, U\}, \quad 3 : \{S, T, U\},$$
$$4 : \{S, V\}, \quad 5 : \{S, T, V\}, \quad 6 : \{S, U, V\}, \quad 7 : \{S, T, U, V\}.$$



On peut interpréter ainsi les états des deux automates : pour savoir si une chaîne α est dans le langage ou non, il suffit de la lire caractère par caractère en retenant les trois derniers caractères lus ; ceux-ci forment l'écriture binaire d'un entier compris entre 0 et 7 ; les états acceptants étant ceux entre 4 et 7, la fonction de transition peut s'écrire

$$\tau(k, N) = (k + 2N) \bmod 8.$$

10.1 Langages réguliers

Commençons par définir des opérations sur les langages. On appelle *concaténation* des langages $\mathcal{L}_1$ et $\mathcal{L}_2$, et on note $\mathcal{L}_1\mathcal{L}_2$ le langage formé des concaténés $\alpha_1\alpha_2$, où α_1 est un mot de $\mathcal{L}_1$ et α_2 est un mot de $\mathcal{L}_2$. La *réunion* $\mathcal{L}_1 \cup \mathcal{L}_2$ de deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ est simplement leur union au sens ensembliste. Enfin, la *fermeture de Kleene* $\mathcal{L}^*$ du langage $\mathcal{L}$ est formée de tous les mots obtenus par concaténation d'un nombre fini, éventuellement nul, de mots de $\mathcal{L}$. Remarquons que, si l'on identifie l'alphabet $\mathcal{A}$ au langage formé des mots de longueur 1, le monoï de libre $\mathcal{A}^*$ coï ncide bien avec la fermeture de Kleene de $\mathcal{A}$.
Un langage est *fini* s'il contient un nombre fini de mots. La classe des *langages réguliers* est la plus petite classe de langages qui comprend les langages finis et qui est fermée pour les opérations de concaténation, réunion et fermeture de Kleene. Un langage régulier est donc obtenu à partir d'un nombre fini de langages finis par une suite finie d'opérations dy type précédent.
Nous allons voir dans les deux propositions suivantes que la classe des langages réguliers coï ncide précisément avec celle des langages reconnus par automate.

Proposition 10.1. *Si A est un automate fini, le langage reconnu par A est régulier.*

Démonstration : par récurrence sur le nombre d'états de A. On désigne par $\mathcal{L} = \{a, b, \ldots\}$ l'ensemble (fini) des étiquettes des boucles menant de S à lui-même. Si l'automate A n'a

qu'un état, l'état initial S, le langage reconnu est $\mathcal{L}^*$ ou l'ensemble vide, selon que S est acceptant ou non. Dans les deux cas, le langage reconnu est régulier. Supposons maintenant la proposition démontrée pour un automate à n états et que l'automate A a $n+1$ états. Notons S_i, $i \in \{1, \cdots, m\}$, les extrémités des flèches d'origine S distinctes de S et a_i, $i \in \{1, \cdots, m\}$, les étiquettes de ces flèches. De même, notons T_j, $j \in \{1, \cdots, p\}$, les origines des flèches d'extrémité S distinctes de S et b_j, $j \in \{1, \cdots, p\}$, leurs étiquettes. Notons encore A_i l'automate obtenu à partir de A en enlevant l'état S ainsi que toutes les flèches qui en viennent ou y aboutissent et en choisissant S_i comme nouvel état initial. Finalement, notons A_{ij} l'automate obtenu à partir de A_i en désignant T_j comme unique état acceptant. D'après l'hypothèse de récurrence, les langages $\mathcal{L}_i$ et $\mathcal{L}_{ij}$ reconnus respectivement par A_i et A_{ij} sont réguliers. Un chemin accepté par A part de S et y revient un certain nombre de fois, puis ne passe plus par S. On voit que le mot associé à un tel chemin appartient à

$$\Big(\bigcup_{i,j} a_i \mathcal{L}_{ij} b_j \cup \mathcal{L}\Big)^* \Big(\bigcup_i a_i \mathcal{L}_i \cup \{\varepsilon\}\Big)$$

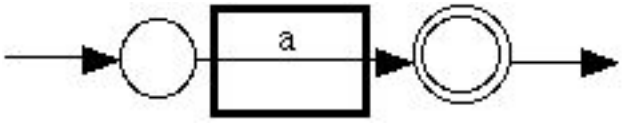
ou à

$$\Big(\bigcup_{i,j} a_i \mathcal{L}_{ij} b_j \cup \mathcal{L}\Big)^* \Big(\bigcup_i a_i \mathcal{L}_i\Big),$$

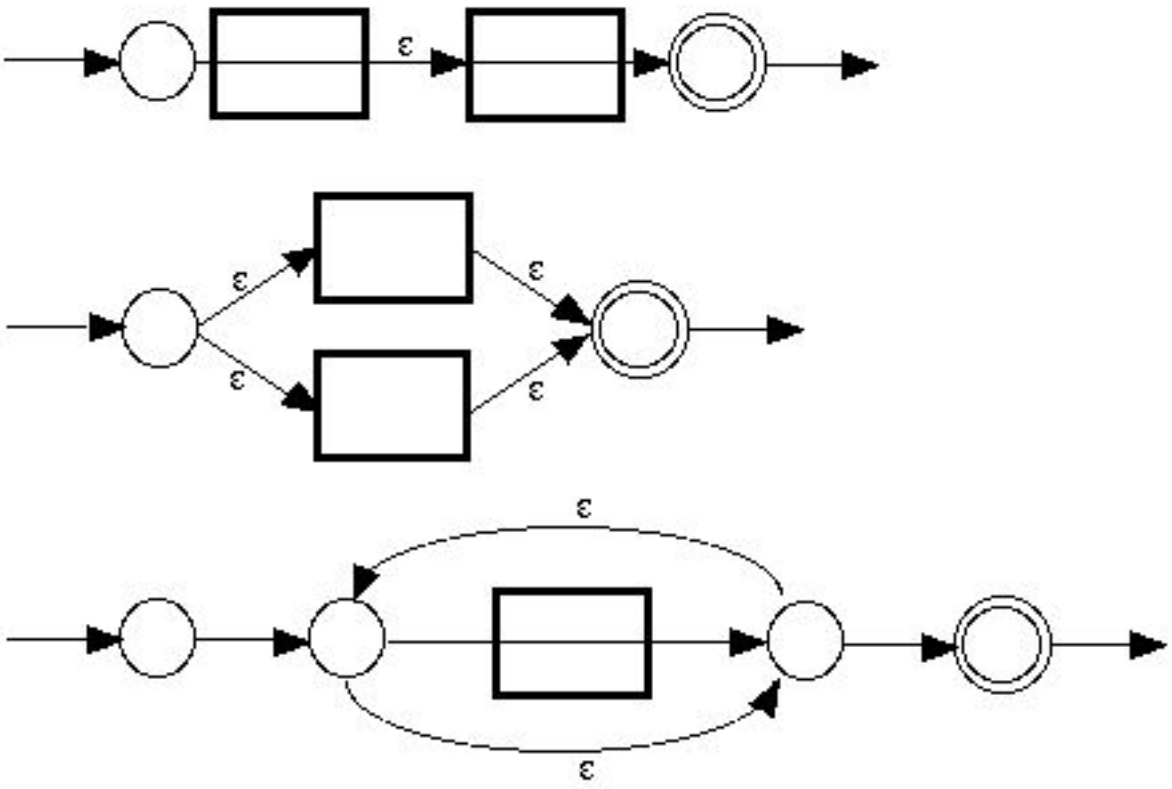
selon que S est acceptant ou non. Dans les deux cas, le langage reconnu par A est encore régulier, ce qui achève la démonstration.

Proposition 10.2. *Pour tout langage régulier non vide, il existe un automate régulier*
(i) qui le reconnaît,
(ii) tel qu'aucune flèche n'a pour extrémité l'état initial,
(iii) tel qu'aucune flèche n'a pour origine l'unique état acceptant.

Nous représenterons un automate satisfaisant (ii) et (iii) par une boî te avec une flèche rentrant à gauche et une flèche sortant à droite. Par exemple, le schéma suivant



décrit l'automate dont le langage associé est formé de l'unique mot a de longueur 1. Les opérations de concaténation, de réunion et de fermeture de Kleene sont représentées respectivement par les schémas suivants :



Comme les langages finis peuvent s'obtenir par les opérations de concaténation et de réunion à partir de langages formés d'un seul mot de longueur 1, on en déduit que la classe des langages reconnus par des automates finis contient tous les langages réguliers.
Une autre façon de représenter les automates non déterministes finis utilise les grammaires. à chaque état de l'automate, on associe un symbole non terminal, l'état initial devenant l'axiome. à chaque transition de X vers Y étiquetée a (respectivement ε), on associe la production $X \to aY$ (respectivement $X \to Y$). Enfin, pour chaque état acceptant X, on écrit la production $X \to \varepsilon$. Une grammaire qui ne contient que des productions des types précédents est appellé une *grammaire régulière*, et on peut donc résumer cette chapter par le

Théorème 10.3. *Un langage est régulier si et seulement si il peut être décrit par une grammaire régulière.*

10.2 Analyse par descente récursive

On a vu que les langages réguliers étaient conceptuellement simples et que la reconnaissance de tels langages était facile à implémenter. Ces techniques conviennent, comme expliqué précédemment, à l'analyse lexicale. Malheureusement, les langages réguliers ne sont pas assez expressifs pour les besoins des langages informatiques. Par exemple, on a la

Proposition 10.4. *Le langage engendré par la grammaire*

$$S \to (S)|\varepsilon$$

n'est pas régulier.

Démonstration : Il est clair que les mots du langage sont formés d'un certain nombre de parenthèses ouvrantes suivi du même nombre de parenthèses fermantes. Supposons qu'il existe

un automate déterministe fini A reconnaissant ce langage, et notons S_n l'état de A atteint à partir de l'état initial après lecture de n parenthèses ouvrantes. Pour $m \neq n$, en partant de S_n et en lisant m parenthèses fermantes, on doit atteindre un état non acceptant alors que la même opération à partir de S_m mène à un état acceptant. On en déduit $S_m \neq S_m$ et l'application : $n \mapsto S_n$ est une injection de l'ensemble des entiers naturels dans l'ensemble des états de A. La contradiction vient du fait que ce dernier ensemble est fini.
Nous allons donc décrire une méthode d'analyse qui s'applique à une classe de grammaires un peu plus générale. L'analyse syntaxique est l'opération qui, pour toute chaîne terminale α, permet de préciser si elle fait partie du langage et surtout, quand c'est le cas, de donner une dérivation explicite menant de l'axiome à la chaîne α. Cette dérivation est rarement unique, mais nous dirons que la grammaire étudiée est *non ambiguë* si toutes les dérivations possibles donnent le même *diagramme syntaxique* : il s'agit d'un arbre dont la racine est l'axiome et où chaque nœ ud est indexé par un symbole, chaque dérivation élémentaire étant représentée par des branches situées au-dessous [2]Tradionnellement, en informatique, les arbres sont représentés racines en l'air et feuilles en bas... d'un nœ ud associé à une symbole non terminal, les feuilles de l'arbre étant associées à des symboles non terminaux ou à ε. Un exemple donnera une idée de ce dont il s'agit. Nous considérons la grammaire

$$\begin{array}{ll} E & \to FMP \\ P & \to +E|\varepsilon \\ M & \to *FM|\varepsilon \\ F & \to 0|1|2|3|4|5|6|7|8|9|(E) \end{array}$$

et la chaîne

$$(4+8)*6+3*5.$$

Le diagramme suivant représente de façon évidente toutes les dérivations possibles de cette chaîne à partir de l'axiome E :
Par exemple,

$$E \to FMP \to (E)MP \to (FMP)MP \to (4MP)MP \to (4P)MP \to (4+E)MP \to \cdots$$

ou

$$E \to FMP \to F*FMP \to F*FM+E \to F*FM+FMP \to (E)*FM+FMP \to \cdots$$

ou encore

$$E \to FMP \to FM+E \to (E)M+E \to (E)*FM+E \to (E)*6M+E \to (E)*6+E \to \cdots$$

La première de ces dérivations, où l'on a choisi à chaque fois de dériver le non terminal le plus à gauche de la chaîne, est appelé *dérivation gauche*. Il y a donc une correspondance biunivoque entre dérivation gauche et diagramme syntaxique.
Le rôle de l'analyseur syntaxique est, à partir d'un flot de symboles fourni par l'analyseur lexical, de déterminer la dérivation gauche menant de l'axiome à ce flot quand elle existe. Dans une analyse par descente récursive, il y a un objectif courant et un symbole terminal courant. L'objectif courant, qui est toujours un symbole, est au départ l'axiome. Pour remplir l'objectif courant, deux cas se présentent :

2. *

• si c'est un symbole terminal a, il s'agit de vérifier que le symbole terminal courant, fourni par l'analyseur syntaxique, est bien égal à a. Si c'est le cas, on passe à l'objectif suivant et au symbole suivant dans le flot ; dans le cas contraire, on conclut que la chaîne originale n'appartient pas au langage ;
• si c'est un symbole non terminal X, on détermine, à l'aide de la valeur du symbole terminal courant laquelle des productions dont le membre de gauche est X doit être appliquée. Les symboles formant le membre de droite de la production choisie deviennent alors des objectifs secondaires dont la résolution successive représentera celle de l'objectif initial.
Dans l'exemple précédent, les symboles fournis par l'analyseur lexical sont dans cet ordre (, 4, +, 8,), ∗, 6, +, 3, ∗ et 5. L'objectif est d'abord de reconnaître un E. La seule production possible étant : $E \to FMP$, le nouvel objectif devient donc de reconnaître successivement un F, un M et un P. Pour reconnaître un F, le symbole terminal courant étant une (, la production choisie est : $F \to (E)$. Il faut donc d'abord reconnaître une parenthèse ouvrante, ce qui se fait simplement en passant au symbole terminal suivant qui est un 4. Le E se décompose de nouveau en FMP, la reconnaissance du F consomme le symbole 4 et il faut maintenant reconnaître un M alors que le symbole terminal courant est un +. Ceci impose le choix de la production : $M \to \varepsilon$. Ensuite, il faut reconnaître le P qui devient $+E$, et ainsi de suite.

10.3 Grammaires LL(1)

La méthode décrite dans la chapter précédente ne fonctionne malheureusement pas pour toutes les grammaires non contextuelles, puisqu'elle repose sur le fait que la connaissance d'un seul symbole terminal courant permet de choisir la production à appliquer. Nous allons déterminer précisément pour quelles grammaires cela est possible et nous expliquerons par quelles techniques on peut étendre la méthode d'analyse par descente récursive à une classe de langages suffisamment large pour les applications envisagées.
Soit donc $G = (\mathcal{N}, \mathcal{T}, \mathcal{S}, \mathcal{P})$ une grammaire non contextuelle, dont nous supposerons qu'elle est réduite. Commençons par dresser la liste des symboles non terminaux dont il est possible de dériver la chaîne ε. On peut y arriver par l'algorithme suivant :

Algorithme 11. $\mathcal{L} \leftarrow \emptyset$ *Répéter* $\mathcal{L}' \leftarrow \mathcal{L}$ *Pour chaque production* $X \to \alpha$, *si* $\alpha \in \mathcal{L}^*$, *ajouter* X *à* $\mathcal{L}$, *fin pour jusqu'à ce que* $\mathcal{L}' = \mathcal{L}$.

à ce point, L contient la liste cherchée. En outre, il est clair que, pour toute chaîne α appartenant à $(\mathcal{N} \cup \mathcal{T})^*$, la chaîne ε dérive de α — ce que nous noterons $V(\alpha)$ — si et seulement si α appartient à $\mathcal{L}^*$.

11 Interpréteur de formules

Le but de ce chapitre est de donner les outils informatiques pour manipuler des formules.

11.1 Grammaire LL(1)

Dans cette section, nous allons étudier des grammaires telles qu'il nous sera facile de faire un programme qui nous dira si un texte fait partie d'un langage. Le programme sera écrit en `C++` , s'arrêtera *via* la fonction `exit(1)` du système si le texte ne fait pas partie du langage.

Dans un premier temps, nous allons prendre comme exemple la grammaire des expressions

$$\begin{array}{rcl} E & = & T \mid T' +' E \\ T & = & F \mid F' *' T \\ F & = & c \mid '(' E ')' \end{array}$$

Où les symbole non-terminaux de la grammaire sont E qui est une expression, T qui est un terme, F qui est un facteur. Ces trois non terminaux qui sont définis par les trois règles. Et où les trois symboles terminaux de la grammaire sont c qui représente un nombre et le symbole $'('$, resp. $')'$ qui représente le caractère (, resp.) .
L'idée est très simple : nous allons supposer que nous disposons d'un analyseur lexical qui découpe l'entrée standard (le texte) en symboles terminaux, qui lit le symbole terminal suivant si le terminal est reconnu. Les symboles terminaux sont numérotés *via* un entier, les caractères ont leur code ASCII et les nombres sont définis par le numéro `c=257` et la fin de fichier par le numéro `EOF = -1`.
Cet analyseur lexical est modélisé par une classe `Lexical` composée de :

- `sym` le symbole courant ;
- `void Nextsym()` fonction qui lit le symbole terminal suivant qui peut être ici `'+' '*' '(' ')' c EOF` ;
- `void Match(int c)` : si le symbole courant est `c`, la fonction lit le symbole suivant, sinon elle génére une erreur ;
- `bool IFSym(int c)` retourne faux si le symbole courant n'est pas `c`, sinon elle lit le symbole suivant et retourne vrai.

Listing 18: *(la classe Lexical)*

```
class Lexical {                                                    //
  typedef double R;
  int sym;
  static const int c=257;                          //    numéro du terminal nombre
  static const int EOF=-1;                           //    numéro du terminal EOF
  istream & cin;                                       //    flot de lecture
  R valeur;                                          //    Valeur du nombre
 public:
   Lexical(istream & s) :cin(s)          //    le constructeur définit le istream
     {NextSym();}                                    //    et lit un symbole

   void Error(const char * message)
    { cerr <<  message  << endl;
      exit(1);}

   void Nextsym() {                                  //    analyse lexicale
    int s=cin.peek();
    if (s==EOF) { sym=EOF; }
    else if (isdigit(s) || s == '.')
            { sym=s; cin >> valeur; assert(cin.good());}
    else if (s=='+' || s=='*' || s=='(' || s==')'  || s==EOF)
            { sym=cin.get(); }
    else if (isspace(s))
           { sym=' '; cin.get(); }
    else
```

```
          { cerr << " caract{è}re invalide " << char(s) << " " << s << endl;
            exit(1);}
  };

  bool Match(int s)                        //    assure que le symbole courant est s
   { if(s!=sym) { cerr << " On attendait le symbole ";
                  if ( s == c) cerr << " un nombre \n" ;
                  if (s == EOF) cerr << "EOF\n";
                  else cerr <<  char(s) << endl;
                  exit(1);
                   }
    else NextSym();
    return true;}

  bool IFSym(int s)                          //     est si le symbole courant est s
    { if (s==sym) { NextSym();return true;} else return false;}

  bool Match(bool b,const char *s)                                          //
    { if (!b) { cerr << " Erreur " << s << " is false " << endl;
                exit(1);}
      return b;
    }
};
```

Une fois le problème de l'analyse lexicale résolu, nous allons nous occuper de la grammaire.
À chaque non-terminal nous allons associer une fonction booléenne qui retourne vrai si l'on a trouvé le non-terminal, faux sinon. Nous utiliserons la fonction `IFSym(c)` pour tester les terminaux. De plus, nous ne voulons pas faire de retour en arrière (un symbole d'avance, le 1 de `LL(1)`) donc dans une concaténation AB si A est vrai et B est faux, il faut remettre l'état du système avant l'appel de A, ce que l'on ne sait pas faire généralement. Donc, dans ce cas, nous générons une erreur.
Il suffit de programmer chaque fonction non-terminale, en factorisant à gauche par non-terminaux, de façon à factoriser tous les termes en commun dans les opération de concaténation.
Les concaténations $E = AB$, $F = Ac$, $G = cA$ sont programmées comme suit :

```
  bool E(){
     if (  A() )
      return Match(B(),'B()');}                    //    Erreur si B() est faux
  bool F(){
     if (  A() )
      return Match(c); }                             //    Erreur si c'est faux
  bool G(){
     if (IFSym(c))
         return Match(A(),"A()");}                 //    Erreur si A() est faux
```

L'opérateur ou dans $P = C \mid c \mid '('$ donne

```
  bool  P() {
   if (C()) return true;
   else if(IFsym(c)) return true;
```

```
   else if(IFsym('(')) return true;
   else return false;}
```

l'opérateur ou dans $Q = C|c|\ '('\ |\varepsilon$ où ε la chaine vide, donne

```
  bool Q() {
   if (C()) return true;
   else if(IFsym(c)) return true;
   else if(IFsym('(')) return true;
   else return true; }                                  //    la chaîne vide rend vrai
```

La règle $E = T|T\ '+'\ E$ se réécrit comme suit $E = T('+'E|\varepsilon)$, où les parenthèses () changent la priorité entre l'opérateur ou et la concaténation. Attention, il ne faut pas confondre avec '(' et ')' qui sont les deux symboles terminaux parenthèses ouvrante et fermante,

$$\begin{array}{rcl} E & = & T|T\ '+'\ E \\ T & = & F|F\ '*'\ T \\ F & = & c|\ '('\ E\ ')' \end{array} \quad \text{en} \quad \begin{array}{rcl} F & = & c|\ '('E\ ')' \\ E & = & T(\ '+'\ E|\varepsilon) \\ F & = & c|\ '('E\ ')' \end{array}$$

Après cette modification élémentaire, il est trivial d'écrire le programme C++ suivant :

Listing 19: *(la classe Grammaire)*

```
class Grammaire :public Lexical {
Grammaire(istream & cc) : Lexical(cc) {}
bool E() { if (T())
              if ( IFSym('+') return Match(E(),"E()");
              else return true;
           else return false; }

bool T() { if(F())
              if ( IFSym('*')  return Match(T(),"T()");
              else return true;
           else return false; }

bool F() {  if( IFSym(c) return true;
            else if ( IFSym('(') { Match(c);return Match(')');}
            else return false; }`
};

int main(int argc,char ** argv) {
 Grammaire exp(cin);
 bool r=exp.E();
 exp.Match(Exp:EOF);                          //    vérification de fin de text Ô
 cout << " Bien dans le langague " << endl;
 return 0;
};
```

Malheureusement, cette technique ne marche pas dans tous les cas, il faut donc encore travailler.

Remarque 13. *La remarque fondamentale, qui donne le principe de fonctionnement et qui définit les grammaires `LL(1)` est : si une règle de production commence par un symbole s, alors la fonction associée retourne vrai ou génère une erreur si le symbole courant est s.*

Nous en déduisons la règle suivante :

Règle 6. Si deux expressions grammaticales commençent par un terminal commun dans une opération logique du type | (ou), alors la grammaire n'est pas `LL(1)`.

Effectivement, la première expression retourne vrai, ou génère une erreur, donc la seconde expression n'est jamais utilisée.
L'autre règle est simplement sur l'utilisation de la règle vide (ε).

Règle 7. La règle vide (ε) doit être mise en fin de chaîne de l'opérateur logique | (ou), car l'expression associée à la règle vide est toujours vraie. Pratiquement, si l'on teste les autres branches du |, il faut que cela soit la dernière régle.

Maintenant nous pouvons définir une grammaire `LL(1)`.
Une grammaire est dite `LL(1)` si le langage reconnu par le programme est le même que celui défini par la grammaire.

Exemple de grammaire non-$LL(1)$

$$\begin{array}{rcl} E & = & T|T\ '+'\ E \\ T & = & F|F\ '*'\ T \\ F & = & P|\ '('\ E\ ')' \\ P & = & c|\ '('\ c\ ','\ c\ ')' \end{array}$$

où c et respectivement $'('\ c\ ','\ c\ ')'$ représentent un `double`, respectivement `complex<double>`.

Exercice 17. Écrire une autre grammaire équivalente et qui est $LL(1)$.

Mais une grammaire sans sémantique n'est pas vraiment utile.
Voici deux versions de la même grammaire avec l'évaluation des calculs. Le principe est d'effectuer les calculs seulement dans le cas où la fonction retourne vrai. Mais attention, deux grammaires qui définissent le même langage peuvent avoir deux sémantiques différentes :

$$\begin{array}{rcl} E & = & T|T+E \\ T & = & F|F*T \\ F & = & c|(E) \end{array} \quad \text{et} \quad \begin{array}{rcl} E & = & T|T+E \\ T & = & F|F*E \\ F & = & c|(E) \end{array}$$

Effectivement, la phrase $2*1+1$ donnera $\{2*1\}+1=3$ pour la grammaire de gauche et $2*\{1+1\}=4$ pour la grammaire de droite.

11.1.1 Une calculette complete

Afin d'écrire une calculette complète qui calcule la valeur associé, Nous remarquons le problème suivant lié à la sémentique et non à la syntaxe.
La grammaire de la calculette devrait etre :

$$\begin{array}{rcl} E & = & T|T\ '+'\ E|T\ '-'\ E \\ T & = & F|F\ '*'\ T|F\ '/'\ T \\ F & = & c|(E) \end{array}$$

Mais cette grammaire faite une associativité à gauche, c'est dire que l'évaluation de $1-1+2$ est donné pas $1-(1+2)$. donc pour corrigé ce problème il suffit d'ajouter deux règles E^- et $T^/$ qui correspondent aux expressions commençant par $-$ ou aux termes commençant par $/$.

$$\begin{array}{rcl} E & = & T|T\ '+'\ E|T\ '-'\ E^- \\ T & = & F|F\ '*'\ T|F\ '/'\ T^/ \\ E^- & = & T|T\ '+'\ E^-|T\ '-'\ E \\ T^/ & = & F|F\ '*'\ T^/|F\ '/'\ T \\ F & = & c|(E) \end{array}$$

Cette écriture est lourde et toutes boucles se font par récurrence, mais si l'on introduit un nouvelle opérateur $*$ dans la description des grammaires qui dit expression suivante doit être là n fois ($n \in \mathbb{N}$) comme dans les expressions régulières, alors nous pouvons réécrit la grammaire comme :

$$\begin{array}{rcl} E & = & T\ *(\ '+'\ T\ |\ '-'\ T\ ) \\ T & = & F\ *(\ '*'\ F\ |\ '/'\ F\ ) \\ F & = & c|(E) \end{array}$$

Voila un exemple complète de calculette, avec la fonction cos, la constante π et trois variables x, y, r qui sont utiliser via les pointeurs `xxxx`, `yyyy`, `rrrrr`.
Le source de l'exemple suivante est disponible à : `http://www.ann.jussieu.fr/~hecht/ftp/Calculette.cpp`

Listing 20: *(Calculette.cpp)*

```
#include <fstream>
#include <iostream>
#include <cstdio>
#include <sstream>
#include <cmath>
#include <cstdlib>
#include <cassert>
#include <cstring>

                          //   #define NDEBUG // uncomment when debugging is over
using namespace std;

//    la grammaire des expression:
//    E = T * ( '+' T | '-' T)
//    T = F * ( '*' F | '/' F )
```

```
//    F = c | '(' E ')' | FUNC1 '(' E ')' | GVAR | '-' F | '+' F;
//    Les fonctions FUNC1 implementées: cos,


double *xxxx,*yyyy,*rrrr;
const double PI=4*atan(1.0);

class Exp {
   public:
                typedef double R;
   private:
            int sym;                              //    la valeur de symbole courant
            static const int Number=257;
            static const int FUNC1=258;
            static const int GVAR=259;
            istringstream  cin;          //    contient la chaine de l'expression

  R valeur, valeuro;                                          //    si sym == Number
  R (*f1)(R), (*f1o)(R);                                       //    si sym == FUNC1
  R *pv, *pvo;                                                  //    Si sym == GVAR

  public:

  Exp(char* f) : cin(f) { cin.seekg(0);                       //    initialisation:
                          NextSym();}                  //    lit le premier symbol
                             //    car l'analyseur a toujours un symbole d'avance

  void NextSym() {                                                //    ReadSymbole
    pvo=pv;                           //    on sauve les valeurs associer au symbol
    f1o=f1;
    valeuro=valeur;

  int c=cin.peek();
   if (isdigit(c) || c == '.') { sym=Number; cin >> valeur;}
   else if (c=='+' || c=='*' || c=='(' || c==')'  || c==EOF) {sym=c;cin.get();}
   else if (c=='-' || c=='/' || c=='^' ) {sym=c;cin.get();}
   else if (isspace(c)) {sym=' '; cin.get();}
   else if (c==EOF ) sym=EOF;
   else if (isalpha(c)) {
     string buf;
     buf += cin.get();
     while (isalnum(cin.peek()))
       buf += cin.get();
     if(buf=="cos") { sym=FUNC1; f1=cos;}
     else if (buf=="x") { sym=GVAR; pv=xxxx;}
     else if (buf=="y") { sym=GVAR; pv=yyyy;}
     else if (buf=="r") { sym=GVAR; pv=rrrr;}
     else if (buf=="pi") { sym=Number; valeur=PI;}
     else { cerr << " mot inconnu '" << buf <<"'" << endl; exit(1);}
       cout << "Id " << buf << " " << sym << endl;}
   else {  cerr << "caractere invalide " << char(c) << " " << c << endl; exit(1);}
  }

 //    ---------------------------------------------------------------------
  void Match(int c) {                          //    le symbole courant doit etre c
       if(c!=sym) { cerr << " On attendait le symbole ";
       if ( c == Number) cerr << " un nombre " ;
```

```
      else cerr << "'"<<char(c)<<"'"; exit(1);}
    else NextSym();
  }

  //    ---------------------------------------------------------------------

  bool IFSym(int c)              //    retourne vraie si le symbole courant est c
    { if (c==sym) { NextSym();return true;}
       else return false;}
 //    ---------------------------------------------------------------------

                        //    les fonctions boolean de la grammaire qui retournent
                              //    la valeur de l'expression dans le paramètre v
bool F(R & v) {
  if(IFSym(Number)) {v=valeuro;    cout << " F " << v <<endl; return true;}
  if(IFSym('-')) {
      if (!F(v) ) { cerr << " -F On attendait un facteur " << endl;exit(1);}
        v = -v;
        return true;
      }
  if(IFSym('+')) {
      if (!F(v) ) { cerr << " +F On attendait un facteur " << endl;exit(1);}
        return true;
      } else if (IFSym('(')) {
          if (!E(v) ) { cerr << " (E) : On attendait un expression"<< endl;exit(1);}
          Match(')');    cout << " (E) " << v <<endl;
          return true;}
  else if (IFSym(GVAR)) {
        v = *pvo;
      return true;
    }
  else if (IFSym(FUNC1))
    {
        R (*ff)(R) = f1o;
      Match('(');
        if (!E(v) ) { cerr << " On attendait un expression " << endl;exit(1);}
      v=ff(v);
        Match(')');    cout << " E : f( exp) =  " << v <<endl;
      return true;
    }
    else return false;
  }

 //    ---------------------------------------------------------------------

bool T(R & v) {
      if (! F(v) ) return false;
      while (1)                      //    correction 1 (operateur gramamtical *)
      {
        if (IFSym('*')) { R vv;
           if (!F(vv) ) { cerr << " On attendait un facteur " << endl;exit(1);}
             v*=vv;}
        else if (IFSym('/')) { R vv;
           if (!F(vv) ) { cerr << " On attendait un facteur " << endl;exit(1);}
             v/=vv;}
        else break;
      }
```

```
       cout << " T " << v <<endl;
      return true;
}

 //    ------------------------------------------------------------------------

bool E(R & v) {
    if (!T(v) ) {  cout << " E false " << endl;  return false;}
      while(1)                        //    correction 2 (operateur gramamtical *)
        if (IFSym('+')) { R vv;
           if (!T(vv) ) { cerr << " On attendait un term " << endl;exit(1);}
           v+=vv;}
           else if (IFSym('-')) { R vv;
             if (!T(vv) ) { cerr << " On attendait un term " << endl;exit(1);}
               v-=vv;}
             else break;
     cout << " E " << v <<endl;
    return true;
}


};

 //    ------------------------------------------------------------------------

int main(int argc,char **argv)
{
   if(argc <2) { cout << " usage : "<< argv[0] << " '1+5*cos(pi) '" <<endl;
   return 1;}

  assert(argc==2);
  Exp exp(argv[1]);
  Exp::R v;         //    pour declare une variable de type real de la calculette
  exp.E(v);
  cout << argv[1]<<  "= "<< v << endl;
  assert(exp.IFSym(EOF));
  return 0;
}
```

11.2 Algèbre de fonctions

Bien souvent, nous aimerions pouvoir considérer des fonctions comme des données classiques, afin de construire une nouvelle fonction en utilisant les opérations classiques `+,-,*,/` ... Par exemple, la fonction $f(x,y) = cos(x) + sin(y)$, pour que la calculette généré une peusdo fonction que l'on pourra évaluer plusieurs fois sans réinterpréter la chaîne de caractère définissant l'expression.
Dans un premier temps, nous allons voir comment l'on peut construire et manipuler en `C++` l'algèbre des fonctions $\mathcal{C}^\infty$, définies sur $\mathbb{R}$ à valeurs dans $\mathbb{R}$.

11.2.1 Version de base

Une fonction est modélisée par une classe (`Cvirt`) qui a juste l'opérateur `()()` virtuel.

Pour chaque type de fonction, on construira une classe qui dérivera de la classe `Cvirt`. Comme tous les opérateurs doivent être définis dans une classe, une fonction sera définie par une classe `Cfonc` qui contient un pointeur sur une fonction virtuelle `Cvirt`. Cette classe contiendra l'opérateur « fonction » d'évaluation, ainsi qu'une classe pour construire les opérateurs binaires classiques. Cette dernière classe contiendra deux pointeurs sur des `Cvirt` qui correspondent aux membres de droite et gauche de l'opérateur et la fonction de $\mathbb{R}^2$ à valeur de $\mathbb{R}$ pour définir l'opérateur.
Les programmes sources sont accessibles à l'adresse :
`http://www.ann.jussieu.fr/~hecht/ftp/cpp/lg/fonctionsimple.cpp`.

Listing 21: *(fonctionsimple.cpp)*

```
 #include <iostream>
                                                    //    pour la fonction pow
#include <math.h>
 typedef double R;
 class Cvirt { public: virtual R operator()(R ) const =0;};

 class Cfonc : public Cvirt { public:
   R (*f)(R);                                      //    pointeur sur la fonction Cpp
   R operator()(R x) const { return (*f)(x);}
   Cfonc( R (*ff)(R)) : f(ff) {} };

 class Cconst : public Cvirt { public:
   R a;
   R operator()(R ) const { return a;}
   Cconst( R aa) : a(aa) {}          };

 class Coper : public Cvirt { public:
   const  Cvirt *g, *d;
   R (*op)(R,R);
   R operator()(R x) const { return (*op)((*g)(x),(*d)(x));}
   Coper( R (*opp)(R,R), const Cvirt *gg, const Cvirt *dd)
         : op(opp),g(gg),d(dd) {}
   ~Coper(){delete g,delete d;}    };

                                                   //    les cinq opérateurs binaires
 static  R Add(R a,R b) {return a+b;}
 static  R Sub(R a,R b) {return a-b;}
 static  R Mul(R a,R b) {return a*b;}
 static  R Div(R a,R b) {return a/b;}
 static  R Pow(R a,R b) {return pow(a,b);}

 class Fonction { public:
   const Cvirt *f;
   R operator()(const R x){ return (*f)(x);}
   Fonction(const Cvirt *ff) : f(ff) {}
   Fonction(R (*ff)(R)) : f(new Cfonc(ff)) {}
   Fonction(R a) : f(new Cconst(a)) {}
   operator const  Cvirt * () const {return f;}
   Fonction operator+(const Fonction & dd) const
        {return new Coper(Add,f,dd.f);}
   Fonction operator-(const Fonction & dd) const
        {return new Coper(Sub,f,dd.f);}
```

```
   Fonction operator*(const Fonction & dd) const
        {return new Coper(Mul,f,dd.f);}
   Fonction operator/(const Fonction & dd) const
        {return new Coper(Div,f,dd.f);}
   Fonction operator^(const Fonction & dd) const
        {return new Coper(Pow,f,dd.f);}
 };

using namespace std;                                    //   introduces namespace std

R Identite(R x){ return x;}
int main()
{
  Fonction Cos(cos),Sin(sin),x(Identite);
  Fonction f((Cos^2)+Sin*Sin+(x*4));          //    attention ^ n'est prioritaire
  cout << f(2) << endl;
  cout << (Cos^2)+Sin*Sin+(x*4))(1) << endl;
  return 0;
}
```

Dans cet exemple, la fonction $f = cos^2 + (sin * sin + x^4)$ sera définie par un arbre de classes qui peut être représenté par :

```
f.f= Coper (Add,t1,t2);                 // t1=(cos^2)     +  t2=(sin*sin+x^4)
    t1.f= Coper (Pow,t3,t4);            // t3=(cos)       ^  t4=(2)
    t2.f= Coper (Add,t5,t6);            // t5=(sin*sin)   +  t6=x^4
        t3.f= Ffonc (cos);
        t4.f= Fconst (2.0);
        t5.f= Coper (Mul,t7,t8);        // t7=(sin)       *  t8=(sin)
            t6.f= Coper (Pow,t9,t10);   // t9=(x)         ^  t10=(4)
                t7.f= Ffonc (sin);
                t8.f= Ffonc (sin);
                t9.f= Ffonc (Identite);
                t10.f= Fconst (4.0);
```

11.2.2 Les fonctions C^∞

Maintenant, nous voulons aussi pouvoir dériver les fonctions. Il faut faire attention, car si la classe contient la dérivée de la fonction, il va y avoir implicitement une récursivité infinie, les fonctions étant indéfiniment dérivables. Donc, pour que la classe fonctionne, il faut prévoir un processus à deux niveaux : l'un qui peut construire la fonction dérivée, et l'autre qui évalue la fonction dérivée et qui la construira si nécessaire.
Donc, la classe `CVirt` contient deux fonctions virtuelles :

```
virtual float operator () (float) = 0;
virtual CVirt *de () {return zero;}
```

La première est la fonction d'évaluation et la seconde calcule la fonction dérivée et retourne la fonction nulle par défaut. Bien entendu, nous stockons la fonction dérivée (`CVirt`) qui ne sera construite que si l'on utilise la dérivée de la fonction.
Nous introduisons aussi les fonctions de $\mathbb{R}^2$ à valeurs dans $\mathbb{R}$, qui seront modélisées dans la classe `Fonction2`.
Les programmes sources sont dans les 2 URL

- `http://www.ann.jussieu.fr/~hecht/ftp/cpp/lg/fonction.cpp`
- `http://www.ann.jussieu.fr/~hecht/ftp/cpp/lg/fonction.hpp`

Toutes ces classe sont définies dans le fichier d'en-tête suivant :

Listing 22: *(fonction.hpp)*

```
#ifndef __FONCTION__
#define __FONCTION__

struct CVirt {
  mutable CVirt *md;                    //    le pointeur sur la fonction dérivée
  CVirt () : md (0) {}
  virtual R operator () (R) = 0;
  virtual CVirt *de () {return zero;}
  CVirt *d () {if (md == 0) md = de (); return md;}
  static CVirt *zero;
};

class Fonction {                                    //    Fonction d'une variable
  CVirt *p;
public:
  operator CVirt *() const {return p;}
  Fonction (CVirt *pp) : p(pp) {}
  Fonction (R (*) (R));                  //    Création à partir d'une fonction C
  Fonction (R x);                                   //    Fonction constante
  Fonction (const Fonction& f) : p(f.p) {}                 //   Constructeur par
copie
  Fonction d () {return p->d ();}                                   //    Dérivée
  void setd (Fonction f) {p->md = f;}
  R operator() (R x) {return (*p)(x);}                  //    Valeur en un point
  Fonction operator() (Fonction);                 //    Composition de fonctions
  friend class Fonction2;
  Fonction2 operator() (Fonction2);
  static Fonction monome (R, int);
};

struct CVirt2 {
  CVirt2 ():  md1 (0), md2 (0) {}
  virtual R operator () (R, R) = 0;
  virtual CVirt2 *de1 () {return zero2;}
  virtual CVirt2 *de2 () {return zero2;}
  CVirt2 *md1, *md2;
  CVirt2 *d1 () {if (md1 == 0) md1 = de1 (); return md1;}
  CVirt2 *d2 () {if (md2 == 0) md2 = de2 (); return md2;}
  static CVirt2 *zero2;
};
```

```
class Fonction2 {                                    //    Fonction de deux variables
  CVirt2 *p;
public:
  operator CVirt2 *() const {return p;}
  Fonction2 (CVirt2 *pp) : p(pp) {}
  Fonction2 (R (*) (R, R));                  //    Création à partir d'une fonction C
  Fonction2 (R x);                                          //    Fonction constante
  Fonction2 (const Fonction2& f) : p(f.p) {}          //   Constructeur par copie
  Fonction2 d1 () {return p->d1 ();}
  Fonction2 d2 () {return p->d2 ();}
  void setd (Fonction2 f1, Fonction2 f2) {p->md1 = f1; p->md2 = f2;}
  R operator() (R x, R y) {return (*p)(x, y);}
  friend class Fonction;
  Fonction operator() (Fonction, Fonction);        //    Composition de fonctions
  Fonction2 operator() (Fonction2, Fonction2);
  static Fonction2 monome (R, int, int);
};

extern Fonction Chs,Identity;
extern Fonction2 Add, Sub, Mul, Div, Abscisse, Ordonnee;

inline Fonction operator+ (Fonction f, Fonction g) {return Add(f, g);}
inline Fonction operator- (Fonction f, Fonction g) {return Sub(f, g);}
inline Fonction operator* (Fonction f, Fonction g) {return Mul(f, g);}
inline Fonction operator/ (Fonction f, Fonction g) {return Div(f, g);}
inline Fonction operator- (Fonction f) {return Chs(f);}

inline Fonction2 operator+ (Fonction2 f, Fonction2 g) {return Add(f, g);}
inline Fonction2 operator- (Fonction2, Fonction2 g) {return Sub(f, g);}
inline Fonction2 operator* (Fonction2 f, Fonction2 g) {return Mul(f, g);}
inline Fonction2 operator/ (Fonction2 f, Fonction2 g) {return Div(f, g);}
inline Fonction2 operator- (Fonction2 f) {return Chs(f);}

#endif
```

Maintenant, prenons l'exemple d'une fonction `Monome` qui sera utilisée pour construire les fonctions polynômes. Pour construire effectivement ces fonctions, nous définissons la classe `CMonome` pour modéliser $x \mapsto cx^n$ et la classe `CMonome2` pour modéliser $x \mapsto cx^{n_1}y^{n_2}$.

Listing 23: *(les classes CMonome et CMonome2)*

```
class CMonome : public CVirt {
R c; int n;
public:
CMonome (R cc = 0, int k = 0) : c (cc), n (k) {}
R operator () (R x);                                          //    return cx^n
CVirt *de () {return n? new CMonome (c * n, n - 1) : zero;}
};
Function Function::monome (R x, int k) {return new CMonome (x, k);}

class CMonome2 : public CVirt2 {
R c; int n1, n2;
```

```
public:
CMonome2 (R cc = 0, int k1 = 0, int k2 = 0) : c (cc), n1 (k1), n2 (k2) {}
R operator () (R x, R y);                                    //    return cx^n1 y^n2
CVirt2 *dex () {return n1? new CMonome2 (c * n1, n1 - 1, n2) : zero;}
CVirt2 *dey () {return n2? new CMonome2 (c * n2, n1, n2 - 1) : zero;}
};
Function2 Function2::monome (R x, int k, int l)
{return new CMonome2 (x, k, l);}
```

En utilisant exactement la même technique, nous pouvons construire les classes suivantes :

CFunc une fonction C++ de prototype `R (*)(R)` .

CComp la composition de deux fonctions f, g comme $(x) \to f(g(x))$.

CComb la composition de trois fonctions f, g, h comme $(x) \to f(g(x), h(x))$.

CFunc2 une fonction C++ de prototype `R (*)(R,R)`.

CComp2 la composition de f, g comme $(x, y) \to f(g(x, y))$.

CComb2 la composition de trois fonctions f, g, h comme $(x, y) \to f(g(x, y), h(x, y))$

Pour finir, nous indiquons juste la définition des fonctions globales usuelles :

```
CVirt *CVirt::zero = new CMonome;
CVirt2 *CVirt2::zero = new CMonome2;

Function::Function (R (*f) (R)) : p(new CFunc(f)) {}
Function::Function (R x) : p(new CMonome(x)) {}
Function Function::operator() (Function f) {return new CComp (p, f.p);}
Function2 Function::operator() (Function2 f) {return new CComp2 (p, f.p);}

Function2::Function2 (R (*f) (R, R)) : p(new CFunc2(f)) {}
Function2::Function2 (R x) : p(new CMonome2(x)) {}
Function Function2::operator() (Function f, Function g)
{return new CComb (f.p, g.p, p);}
Function2 Function2::operator() (Function2 f, Function2 g)
{return new CComb2 (f.p, g.p, p);}

static R add (R x, R y) {return x + y;}
static R sub (R x, R y) {return x - y;}

Function Log = new CFunc1(log,new CMonome1(1,-1));
Function Chs = new CMonome (-1., 1);
Function Identity = new CMonome (-1., 1);

Function2 CoordinateX = new CMonome2 (1., 1, 0);                    //    x
Function2 CoordinateY = new CMonome2 (1., 0, 1);                    //    y
Function2 One2 = new CMonome2 (1., 0, 0);                           //    1
Function2 Add = new CFunc2 (add, Function2(1.), Function2(1.));
Function2 Sub = new CFunc2 (sub, Function2(1.), Function2(-1.));
Function2 Mul = new CMonome2 (1., 1, 1);
Function2 Div = new CMonome2 (1., 1, -1);
                                                    //    pow(x,y) = x^y = e^log(x)y
Function2 Pow = new CFunc2 (pow, CoordinateY*Pow(CoordinateX,
CoordinateY-One2)),Log(CoordinateX)*Pow);
```

Avec ces définitions, la construction des fonctions classiques devient très simple ; par exemple, pour construire les fonctions sin, cos il suffit d'écrire :

```
Function Cos(cos),Sin(sin);
Cos.setd(-Sin);                                    //    définit la dérivée de cos
Sin.setd(Cos);                                     //    définit la dérivée de sin
Function d4thCos=Cos.d().d().d().d();        //    construit la dérivée quatrième
```

11.3 Un petit langage

Pour faire un langage, il faut une machine virtuelle car l'on ne connaît pas le langage machine. Un idée simple est d'utiliser l'algèbre de fonctions comme machine virtuelle. Mais dans notre cas les fonctions seront juste des fonctions sans argument.

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef double R;

  static R Add(R a,R b) { return a+b;}
  static R Comma(R a,R b) { return a,b;}
  static R Mul(R a,R b) { return a*b;}
  static R Div(R a,R b) { return a/b;}
  static R Sub(R a,R b) { return a-b;}
  static R Lt(R a,R b) { return a<b;}
  static R Gt(R a,R b) { return a>b;}
  static R Le(R a,R b) { return a<=b;}
  static R Ge(R a,R b) { return a>=b;}
  static R Eq(R a,R b) { return a==b;}
  static R Ne(R a,R b) { return a!=b;}

  static R Neg(R a)     { return -a;}
  static R Not(R a)     { return!a;}
static R Print(R a)     { cout << a << " ";  return a;}

class Exp { public:

  class ExpBase{ public:    virtual R operator()() const = 0;
    virtual ExpBase * set(const ExpBase *) const {return 0;}
  };

  class ExpConst : public ExpBase {public:
    R a;
    ExpConst(R aa) : a(aa) {}
    R operator()() const {return a;}};

  class ExpPString : public  ExpBase {public:
    const string *p;
    ExpPString(const string *pp) :p(pp){}
```

```
  R operator()() const {cout << *p; return 0;}
};

class ExpPtr : public  ExpBase {public:

  class SetExpPtr  : public  ExpBase  {public:
    R *a;
    const ExpBase *e;
    R operator()() const {return *a=(*e)();}                          //    set
    SetExpPtr(R * aa, const ExpBase *ee) : a(aa),e(ee) {}};
  R *a;
  ExpPtr(R *aa) : a(aa) {}
  R operator()() const {return *a;}
  virtual ExpBase * set(const ExpBase * e) const
                {return new SetExpPtr(a,e);}};

class ExpOp2 : public ExpBase {public:
  typedef  R (*func)(R,R);
  const ExpBase &a,&b; func op;
  ExpOp2(func o,const ExpBase &aa,const ExpBase &bb )
         : a(aa),b(bb),op(o) {assert(&a && &b);}
  R operator()() const {R aa=a();R bb=b(); return op(aa,bb);}};

class ExpOp2_ : public ExpBase {public:
  typedef  R (*func)(const ExpBase &,const ExpBase &);
  const ExpBase &a,&b;func op;

  ExpOp2_(func o,const ExpBase &aa,const ExpBase &bb )
         : a(aa),b(bb),op(o) {assert(&a && &b);}
  R operator()() const {return op(a,b);}};


class ExpOp : public ExpBase {public:
  const ExpBase &a; R (*op)(R);
  ExpOp(R (*o)(R),const ExpBase &aa) : a(aa),op(o) {assert(&a);}
  R operator()() const {return op(a());}};


const ExpBase * f;
Exp() : f(0) {}
Exp(const ExpBase *ff) : f(ff) {}
Exp(R a)  : f(  new ExpConst(a)) {}
Exp(R* a) :f( new ExpPtr(a)) {}
R operator()() const {return (*f)();}
operator const ExpBase * () const  {return f;}
Exp operator=(const Exp & a) const { return f->set(a.f);}
Exp operator,(const Exp & a) const { return new ExpOp2(Comma,*f,*a.f);}
Exp operator+(const Exp & a) const { return new ExpOp2(Add,*f,*a.f);}
Exp operator-(const Exp & a) const { return new ExpOp2(Sub,*f,*a.f);}
Exp operator-() const              { return new ExpOp(Neg,*f);}
Exp operator!() const              { return new ExpOp(Not,*f);}
Exp operator+() const              { return *this;}
Exp operator/(const Exp & a) const { return new ExpOp2(Div,*f,*a.f);}
Exp operator*(const Exp & a) const { return new ExpOp2(Mul,*f,*a.f);}

Exp operator<(const Exp & a) const { return new ExpOp2(Lt,*f,*a.f);}
Exp operator>(const Exp & a) const { return new ExpOp2(Gt,*f,*a.f);}
```

```
  Exp operator<=(const Exp & a) const { return new ExpOp2(Le,*f,*a.f);}
  Exp operator>=(const Exp & a) const { return new ExpOp2(Ge,*f,*a.f);}
  Exp operator==(const Exp & a) const { return new ExpOp2(Eq,*f,*a.f);}
  Exp operator!=(const Exp & a) const { return new ExpOp2(Ne,*f,*a.f);}

  Exp comp(R (*ff)(R )){  return new ExpOp(ff,*f);}
  Exp comp(R (*ff)(R,R ),const Exp & a){  return new ExpOp2(ff,*f,*a.f);}

};

Exp comp(R (*ff)(R ),const Exp & a){  return new Exp::ExpOp(ff,*a.f);}
Exp comp(R (*ff)(R,R ),const Exp & a,const Exp & b)
                              {  return new Exp::ExpOp2(ff,*a.f,*b.f);}
Exp comp(Exp::ExpOp2_::func ff,const Exp & a,const Exp & b)
                              {  return new Exp::ExpOp2_(ff,*a.f,*b.f);}
Exp print(const  Exp & a)    { return new Exp::ExpOp(Print,*a.f);}
Exp print(const string * p)  { return new Exp::ExpPString(p);}
R While(const  Exp::ExpBase & a,const  Exp::ExpBase & b) {
  R r=0;
  while( a() ) { r=b();}
  return r;
}

                                    //    ------ utilisation de la stl --------
class TableOfIdentifier {
public:
  typedef  pair<int,void  *> value;                          //    typeid + data
  typedef map<string,value> maptype;
  typedef maptype::iterator iterator;
  typedef maptype::const_iterator const_iterator;
  maptype m;

  value Find(const string & id) const;
  void * Add(const string & id,int i=0,void *p=0)
  { m.insert(make_pair<string,value>(id,value(i,p))); return p;}
  void * Add(const string & id, R (*f)(R));
  void * Add(const string & id,const  string & value);
  const Exp::ExpBase * Add(const string & id,const Exp::ExpBase * f);

} tableId;
```

Remarque dans le code précédent, la table des symboles est juste une `map` de la STL, ou les valeurs sont des paires de `int` le type du symblos et `void *` pointeur sur la valeur du symbole, ici les types des symboles peuvent être de

—

11.3.1 La grammaire en bison ou yacc

Il y a une très belle documentation de bison donner avec la distribution de paquet `bison`, je vous conseille vivement sa lecture, il y a de nombreux exemples. Mais, je pense que le code suivante doit vous permettre de comprendre, sauf pour la priorité des opérateurs qui est gérée par les obscur commandes `%left` et `%right`.

```
%{
#include <iostream>
#include  <complex>
#include <string>
#include <cassert>
using namespace std;
#ifdef __MWERKS__
#include "alloca.h"
#endif
#include "Expr.hpp"

                                                    //    to store le code generer
 const  Exp::ExpBase  * leprogram=0;

inline void yyerror(const char * msg)
{  cerr << msg << endl;
   exit(1);}

int yylex();
%}


%union{
 double dnum;
 string * str;
 const  Exp::ExpBase * exp;
 R (*f1)(R);
 void * p;
}

%token WHILE
%token PRINT
%token GE
%token LE
%token EQ
%token NE

%token <dnum> DNUM
%token <str> NEW_ID
%token <str> STRING
%token <exp> ID
%token <f1> FUNC1
%token ENDOFFILE
%type  <exp> code
%type  <exp> parg
%type  <exp> instr
%type  <exp> expr
%type  <exp> expr1
%type  <exp> start
%type  <exp> unary_expr
%type  <exp> pow_expr
%type  <exp> primary
%left  <exp> ','
%left  <exp>  '<' '>' EQ NE LE GE
%left  <exp>  '+' '-'
%left  <exp> '*' '/'  '%'
%right <exp>  '^'
```

```
%%
start:   code  ENDOFFILE { leprogram=$1; return 0; }
;

code : instr
      | code instr { $$=(Exp($1),$2);}
;

instr:  NEW_ID '=' expr ';' { $$=(Exp(tableId.Add(*$1,Exp(new R)))=$3);
                              cout << "new id = " << (*$$)() << endl;}
     | ID '=' expr ';' { $$ = (Exp($1)=$3);
                         cout << "id = " << (*$$)() << endl;}
     | PRINT parg ';' {$$=$2;}
     | WHILE '(' expr ')' '{' code '}' { $$ = comp(While,Exp($3),Exp($6));}
     | expr ';'
;

parg:   expr { $$=print($1);}
      | STRING { $$=print($1);}
      |  parg ',' parg {$$=(Exp($1),$3);}
;

expr: expr1
    | expr '<' expr {$$ = Exp($1)<Exp($3);}
    | expr '>' expr {$$ = Exp($1)>Exp($3);}
    | expr LE expr {$$ = Exp($1)<=Exp($3);}
    | expr GE expr {$$ = Exp($1)>=Exp($3);}
    | expr EQ expr {$$ = Exp($1)==Exp($3);}
    | expr NE expr {$$ = Exp($1)!=Exp($3);}
;

expr1: unary_expr
    | expr1 '+' expr1 {$$ = Exp($1)+$3;}
    | expr1 '-' expr1 {$$ = Exp($1)-Exp($3);}
    | expr1 '*' expr1 {$$ = Exp($1)*$3;}
    | expr1 '/' expr1 {$$ = Exp($1)/$3;}
;

unary_expr:   pow_expr
  | '-'  pow_expr {$$=-Exp($2);}
  | '!'  pow_expr {$$=!Exp($2);}
  | '+'  pow_expr {$$=$2;}
;

pow_expr: primary
     |      primary  '^' unary_expr   {$$=comp(pow,$1,$3);}
;

primary:   DNUM {$$=Exp($1);}
     | '(' expr ')' { $$=$2;}
     |  ID { $$=$1;}
     |  FUNC1 '('expr ')' { $$=comp($1,$3);}
;

%%
```

```
//    mettre ici les fonctions utilisateurs
//    et généralement le programme principal.
```

11.3.2 Analyseur lexical

Théoriquement, il eut fallu utiliser `flex` pour construire l'analyseur lexical, mais le code est simple donc je l'ai réécris (une page).

```
int yylex() {                    //    remarque yylex est le nom donne par bison.
  L1:
   if (cin.eof())
     return ENDOFFILE;
   int c=cin.peek();
   if (c<0) return
     ENDOFFILE;
   if (isdigit(c)) {                              //    un nombre le C++ va le lire
       cin >> yylval.dnum; assert(cin.eof() || cin.good()); return DNUM;}
   else if (c=='.') {              //    c''est peut etre un nombre ou un point .
             cin.get(); c=cin.peek(); cin.putback('.');
             if (isdigit(c)) {                                 //    c'est un nombre
                   cin >> yylval.dnum;
                   assert(cin.eof() || cin.good());
                   return DNUM;}
             return cin.get();}                               //    c'est un point .
   else if (c=='"')                                  //    une chaine de caractères
     {  yylval.str = new string();
        cin.get();
        while (cin.peek() != '"' && cin.peek() != EOF)
          *yylval.str += char(cin.get());
         assert(cin.peek() == '"' );
         cin.get();
        return STRING;     }
   else if (isalpha(c)) {                                 //    un identicateur
      yylval.str = new string();
      do {
        *yylval.str += char(cin.get());
       } while (isalnum(cin.peek()));
      pair<int,void*> p=tableId.Find(*yylval.str);
     if (p.first==0) return NEW_ID;                              //    n'existe pas
      else {                                  //    existe on retourne type et valeur
         delete yylval.str;  yylval.p=p.second; return p.first;} }
   else if (isspace(c))                                   //    on saute les blancs
     {cin.get(); goto L1;}
   else {                                          //    caractère spéciaux du langage
     c = cin.get();                                    //    le caractère courant
     int cc = cin.peek();                              //    le caractère suivant
     if (c=='+' || c=='*' || c=='(' || c==')' ) return c;
     else if (c=='-' || c=='/' || c=='{' || c=='}' ) return c;
     else if (c=='^' || c==';' || c==',' || c=='.' ) return c;
     else if (cc!= '=' && (c=='<' || c=='>' || c=='=' || cc=='!' ))
               return c;
     else if (c=='<' && cc=='=') { cin.get(); return LE;}
     else if (c=='>' && cc=='=') { cin.get(); return GE;}
     else if (c=='=' && cc=='=') { cin.get(); return EQ;}
     else if (c=='!' && cc=='=') { cin.get(); return NE;}
```

```
     else {                                                          //    erreur
               cerr << " caractere invalide " << char(c) << " " << c << endl;
               exit(1);}   }
}
```

le programme principale :

```
int main (int  argc, char **argv)
{
                                                                  //    yydebug = 1;
  R x,y;

  x=0;
  y=0;

  tableId.Add("x",Exp(&x));
  tableId.Add("y",Exp(&y));
  tableId.Add("cos",cos);
  tableId.Add("end",ENDOFFILE);
  tableId.Add("while",WHILE);
  tableId.Add("print",PRINT);
  tableId.Add("endl","\n");
  yyparse();
  cout << " Compile OK code: " << leprogram <<endl;
  if(leprogram)
    {
      R r= ( * leprogram)();                                //    execution du code
      cout << " return " << r <<endl;
    }

  return 0;
}
```

Les fonctions utiles pour la table des symboles : recherche un symbole `Find` qui retour les deux valeur associer, type et pointeur. Et trois méthodes pour ajouter des symboles de type différent, une chaîne de caractères, une fonction `C++ : double (*)(double)` , ou une expression du langage.

```
 pair<int,void*> TableOfIdentifier::Find(const string & id) const  {
    maptype::const_iterator i=m.find(id);
    if (i == m.end()) return make_pair<int,void*>(0,0);
    else return i->second;
  }

void * TableOfIdentifier::Add(const string & id,const string & value)
  { return Add(id,STRING ,(void*) new string(value));}

void * TableOfIdentifier::Add(const string & id, R (*f)(R))
  { return Add(id,FUNC1,(void*)f);}

const Exp::ExpBase * TableOfIdentifier::Add(const string & id,
                                            const Exp::ExpBase * f)
  {  Add(id,ID,(void*)f); return f;}
```

Voilà un petit programme `example-exp.txt` testé :

```
i=1;
while(i<10)
{
  i = i+1;
  a= 2^i+cos(i*3+2);
  print " i = ",  i,a, endl;
}
i;
end
```

et la sortie :3

```
Brochet:~/work/Cours/InfoBase/l3-lg hecht$ ./exp <example-exp.txt
new id = 1
id = 2
new id = 3.8545
 Compile OK code: 0x3007b0
 i = 2 3.8545
 i = 3 8.00443
 i = 4 16.1367
 i = 5 31.7248
 i = 6 64.4081
 i = 7 127.467
 i = 8 256.647
 i = 9 511.252
 i = 10 1024.83
 return 10
```

Exercice 18.

Ajouter un teste a la grammaire : **if** `(expr) instruction` **else** `instruction` . Pour cela, il faut ajouter un nouveau type `ExpOp3_` de `Exp` en s'aidant de `ExpOp2_` dans le fichier `Expr.hpp`, Puis ajouter les `token` `if` et `else` à bison, ajouter en plus à la définition de `instr` le code correspondant :

```
  | IF '(' expr ')' instr {$$= comp2(If2,Exp($3),Exp($5));}
  | IF '(' expr ')' instr  ELSE instr
                {$$=comp3(If3,Exp($3),Exp($5),Exp($7));}
  |  '{' code '}'  { $$ = $2;}
```

Pour finir, il suffit de créer dans le `main`, les deux nouveau mots clefs (`if`,`else`)comme `while`.

12 Différentiation automatique

Les dérivées d'une fonction décrite par son implémentation numérique peuvent être calculées automatiquement et exactement par ordinateur, en utilisant la différentiation automatique (DA). La fonctionnalité de DA est très utile dans les programmes qui calculent la sensibilité par rapport aux paramètres et, en particulier, dans les programmes d'optimisation et de design.

12.1 Le mode direct

L'idée de la méthode directe est de différencier chaque ligne du code qui définit la fonction. Les différentes méthodes de DA se distinguent essentiellement par l'implémentation de ce principe de base (cf. [**?**]).
L'exemple suivant illustre la mise en œuvre de cette idée simple.

Exemple : Considérons la fonction $J(u)$, donnée par l'expression analytique suivante :

$$J(u) \quad \text{avec} \quad \begin{array}{l} x = u - 1/u \\ y = x + \log(u) \\ J = x + y. \end{array}$$

Nous nous proposons de calculer automatiquement sa dérivée $J'(u)$ par rapport à la variable u, au point $u = 2.3$.

Méthode : Dans le programme de calcul de $J(u)$, chaque ligne de code sera précédée par son expression différentiée (avant et non après à cause des instructions du type $x = 2 * x + 1$) :

$$\begin{array}{l} dx = du + du/(u * u) \\ \mathbf{x = u - 1/u} \\ dy = dx + du/u \\ \mathbf{y = x + \log(u)} \\ dJ = dx + dy \\ \mathbf{J = x + y} \end{array}$$

Ainsi avons nous associé à toute variable (par exemple x) une variable supplémentaire, sa différentielle (dx). La différentielle devient la dérivée seulement une fois qu'on a spécifié la variable de dérivation. La dérivée (dx/du) est obtenue en initialisant toutes les différentielles à zéro en début de programme sauf la différentielle de la variable de dérivation (ex du) que l'on initialise à 1.
La valeur de la dérivée $J'(u)|_{u=2.3}$ est donc obtenue en exécutant le programme ci-dessus avec les valeurs initiales suivantes : $u = 2.3, du = 1$ et $dx = dy = 0$.

Remarque 14. *Les langages C,C++ , FORTRAN... ont la notion de* constante. *Donc si l'on sait que, par exemple, $a = 2$ dans tous le programme et que a ne changera pas, on n'est pas obligé de lui associer une différentielle. Par exemple, la fonction C*

```
float mul(const int a, float u)
{  float x; x=a*u; return x;}
```

se dérive comme suit :

```
float dmul(const int a, float u, float du)
{  float x,dx; dx = a*du; x=a*u; return dx;}
```

Les structures de contrôle (boucles et tests) présentes dans le code de définition de la fonction seront traitées de manière similaire. En effet, une instruction de test de type *if* où a est prédéfini,

```
y = a;
if ( u>0) x = u;
  else    x = 2*u;
J=x+y;
```

peut être vue comme deux programmes distincts :
• le premier calcule

```
y=a; x=u; J=x+y;
```

et avec la DA, il doit retourner

```
dy=0; y=a; dx=du; x=u; dJ=dx+dy; J=x+y;
```

• le deuxième calcule

```
y=a; x=2*u; J=x+y;
```

et avec la DA, il doit retourner

```
dy=0; y=a; dx=2*du; x=2*u; dJ=dx+dy; J=x+y;
```

Les deux programmes sont réunis naturellement sous la forme d'un unique programme

```
dy=0; y=a;
if (u>0) {dx=du; x=u;}
    else {dx=2*du; x=2*u;}
dJ=dx+dy; J=x+y;
```

Le même traitement est appliqué à une structure de type boucle :

```
x=0;
for( int i=1; i<= 3; i++) x=x+i/u;
cout << x << endl;
```

qui, en fait, calcule

```
x=0; x=x+1/u; x=x+2/u; x=x+3/u; cout << x << endl;
```

Pour la DA, il va falloir calculer

```
dx=0;x=0;
dx=dx-du/(u*u); x=x+1/u;
dx=dx-2*du/(u*u); x=x+2/u;
dx=dx-3*du/(u*u); x=x+3/u;
cout << x <<'\t'<< dx << endl;
```

ce qui est réalisé simplement par l'instruction :

```
dx=0;x=0;
for( int i=1; i<= 3; i++)
    { dx=dx-i*du/(u*u); x=x+i/u;}
cout << x <<'\t'<< dx << endl;
```

Limitations :
• Si dans les exemples précédents la variable booléene qui sert de test dans l'instruction `if` et/ou les limites de variation du compteur de la boucle `for` dépendent de u, l'implémentation décrite plus haut n'est plus adaptée. Il faut remarquer que dans ces cas, la fonction définie par ce type de programme n'est plus différentiable par rapport à la variable u, mais est différentiable à droite et à gauche et les dérivées calculées comme ci-dessus sont justes.

Exercice 19. ‖ Ecrire le programme qui dérive une boucle `while`.

• Il existe des fonctions non-différentiables pour des valeurs particulières de la variable (par exemple, $\sqrt{u}$ pour $u = 0$). Dans ce cas, toute tentative de différentiation automatique pour ces valeurs conduit à des erreurs d'exécution du type *overflow* ou *NaN* (not a number).

12.2 Fonctions de plusieurs variables

La méthode de DA reste essentiellement la même quand la fonction dépend de plusieurs variables. Considérons l'application $(u_1, u_2) \to J(u_1, u_2)$ définie par le programme suivant :

$$y_1 = l_1(u_1, u_2) \quad y_2 = l_2(u_1, u_2, y_1) \quad J = l_3(u_1, u_2, y_1, y_2) \tag{31}$$

En utilisant la méthode de DA décrite précédemment, nous obtenons :

$$\begin{aligned}
&dy_1 = \partial_{u_1} l_1(u_1, u_2)dx_1 + \partial_{u_2} l_1(u_1, u_2)dx_2\\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2})\\
&dy_2 = \partial_{u_1} l_2 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_2 dy_1\\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1})\\
&dJ = \partial_{u_1} l_3 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_3 dy_1 + \partial_{y_2} l_3 dy_2\\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2}).
\end{aligned}$$

Pour obtenir dJ pour (u_1, u_2) donnés, il faut exécuter le programme deux fois : une première fois avec $dx_1 = 1, dx_2 = 0$, ensuite, une deuxième fois, avec $dx_1 = 0, dx_2 = 1$.
Une meilleure solution est de dupliquer les lignes $dy_i = \cdots$ et les évaluer successivement pour $dx_i = \delta_{ij}$. Le programme correspondant :

$$
\begin{aligned}
d1y_1 &= \partial_{u_1} l_1(u_1, u_2) d1x_1 + \partial_{u_2} l_1(u_1, u_2) d1x_2 \\
d2y_1 &= \partial_{u_1} l_1(u_1, u_2) d2x_1 + \partial_{u_2} l_1(u_1, u_2) d2x_2 \\
\mathbf{y_1} &= \mathbf{l_1}(\mathbf{u_1}, \mathbf{u_2}) \\
d1y_2 &= \partial_{u_1} l_2 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_2 d1y_1 \\
d2y_2 &= \partial_{u_1} l_2 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_2 d2y_1 \\
\mathbf{y_2} &= \mathbf{l_2}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}) \\
d1J &= \partial_{u_1} l_3 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_3 d1y_1 + \partial_{y_2} l_3 d1y_2 \\
d2J &= \partial_{u_1} l_3 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_3 d2y_1 + \partial_{y_2} l_3 d2y_2 \\
\mathbf{J} &= \mathbf{l_3}(\mathbf{u_1}, \mathbf{u_2}, \mathbf{y_1}, \mathbf{y_2})
\end{aligned}
$$

sera exécuté pour les valeurs initiales : $d1x_1 = 1,\ d1x2 = 0,\ d2x_1 = 0,\ d2x_2 = 1$.

12.3 Une bibliothèque de classes pour le mode direct

Il existe plusieurs implémentations de la différentiation automatique, les plus connues étant **Adol-C, ADIFOR** et **Odyssee**. Leur utilisation implique une période d'apprentissage importante ce qui les rend peu accessibles aux programmeurs débutants. C'est la raison pour laquelle nous présentons ici une implémentation utilisant le mode direct qui est très simple d'utilisation, quoique moins performante que le mode inverse.
On utilise la technique de surcharge d'opérateurs (voir chapitre **??**). Toutefois, cette technique n'est efficace pour le calcul de dérivées partielles que si le nombre de variables de dérivation est inférieur à la cinquantaine. Pour une implémentation similaire en FORTRAN 90, voir Makinen [**?**].

12.4 Principe de programmation

Considérons la même fonction

$$
J(u) \quad \text{avec} \quad \begin{aligned} x &= 2u(u+1) \\ y &= x + \sin(u) \\ J &= x * y. \end{aligned}
$$

Par rapport à la méthode décrite précédemment, nous allons remplacer chaque variable par un tableau de dimension deux, qui va stocker la valeur de la variable et la valeur de sa différentielle. Le programme modifié s'écrit :

```
float y[2],x[2],u[2];
                                                    //    dx = 2 u du + 2 du (u+1)
      x[1] = 2 * u[0] * u[1] + 2 * u[1] * (u[0] + 1);
                                                          //    x = 2 u (u+1)
```

```
x[0] = 2 * u[0] * (u[0] + 1);
y[1] = x[1] + cos(u[0])*u[1];
y[0] = x[0] + sin(u[0]);
J[1] = x[1] * y[0] + x[0] * y[1];
                                                    //     J = x * y
J[0] = x[0] * y[0];
```

L'étape suivante de l'implémentation (voir [**?**], chapitre 4) consiste à créer une classe C++ qui contient comme données membres les tableaux introduits plus haut. Les opérateurs d'algèbre linéaire classiques seront redéfinis à l'intérieur de la classe pour prendre en compte la structure particulière des données membres. Par exemple, les opérateurs d'addition et multiplication sont définis comme suit :

12.5 Implémentation comme bibliothèque C++

Tous les fichiers sont dans l'archive `http://www.ann.jussieu.fr/~hecht/ftp/MN406/FH/autodiff.tar.bz`.
Afin de rendre possible le calcul des dérivées partielles (N variables), l'implémentation C++ de la DA va utiliser la classe suivante :

Listing 24: *(la classe `ddouble`)*

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

struct ddouble {
 double val,dval;
 ddouble(double x,double dx): val(x),dval(dx) {}
 ddouble(double x): val(x),dval(0) {}

};

inline ostream &  operator<<(ostream & f,const  ddouble & a)
{ f << a.val << " ( d = "<< a.dval << ") "; return f;}
inline ddouble operator+(const ddouble & a,const ddouble & b)
   { return ddouble(a.val+b.val,a.dval+b.dval);}
inline ddouble operator+(const double & a,const ddouble & b)
   { return ddouble(a+b.val,b.dval);}

inline ddouble operator*(const ddouble & a,const ddouble & b)
   { return ddouble(a.val*b.val,a.dval*b.val+a.val*b.dval);}
inline ddouble operator*(const double & a,const ddouble & b)
   { return ddouble(a*b.val,a*b.dval);}
inline ddouble operator/(const ddouble & a,const ddouble & b)
   { return ddouble(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
inline ddouble operator/(const double & a,const ddouble & b)
   { return ddouble(a/b.val,(-a*b.dval)/(b.val*b.val));}
inline ddouble operator-(const ddouble & a,const ddouble & b)
   { return ddouble(a.val-b.val,a.dval-b.dval);}

inline ddouble operator-(const ddouble & a)
   { return ddouble(-a.val,-a.dval);}
inline ddouble sin(const ddouble & a)
```

```
   { return ddouble(sin(a.val),a.dval*cos(a.val));}
inline ddouble cos(const ddouble & a)
   { return ddouble(cos(a.val),-a.dval*sin(a.val));}
inline ddouble exp(const ddouble & a)
   { return ddouble(exp(a.val),a.dval*exp(a.val));}
inline ddouble fabs(const ddouble & a)
   { return (a.val > 0)? a : -a;}
inline  bool operator<(const ddouble & a ,const ddouble & b)
   { return a.val < b.val;}
```

---

voila un petit exemple d'utilisation de ces classes

```
template<typename R> R f(R x)
{
   R y=(x*x+1.);
   return y*y;
}

int main()
{
        ddouble x(2,1);
        cout << f(2.0) << " x = 2.0,  (x*x+1)^2 " << endl;
        cout << f(ddouble(2,1)) << "2 (2x) (x*x+1)   " << endl;
        return 0;
}
```

Mais de faite l'utilisation des (templates) permet de faire une utilisation recursive.

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

template<class R> struct Diff {
 R val,dval;
 Diff(R x,R dx): val(x),dval(dx) {}
 Diff(R x): val(x),dval(0) {}
};

template<class R>  ostream &  operator<<(ostream & f,const  Diff<R> & a)
{ f << a.val << " ( d = "<< a.dval << ") "; return f;}
template<class R> Diff<R> operator+(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val+b.val,a.dval+b.dval);}
template<class R> Diff<R> operator+(const R & a,const Diff<R> & b)
   { return Diff<R>(a+b.val,b.dval);}

template<class R> Diff<R> operator*(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val*b.val,a.dval*b.val+a.val*b.dval);}
template<class R> Diff<R> operator*(const R & a,const Diff<R> & b)
   { return Diff<R>(a*b.val,a*b.dval);}
template<class R> Diff<R> operator/(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val/b.val,(a.dval*b.val-a.val*b.dval)/(b.val*b.val));}
template<class R> Diff<R> operator/(const R & a,const Diff<R> & b)
   { return Diff<R>(a/b.val,(-a*b.dval)/(b.val*b.val));}
```

```
template<class R> Diff<R> operator-(const Diff<R> & a,const Diff<R> & b)
   { return Diff<R>(a.val-b.val,a.dval-b.dval);}

template<class R> Diff<R> operator-(const Diff<R> & a)
   { return Diff<R>(-a.val,-a.dval);}
template<class R>Diff<R> sin(const Diff<R> & a)
   { return Diff<R>(sin(a.val),a.dval*cos(a.val));}
template<class R> Diff<R> cos(const Diff<R> & a)
   { return Diff<R>(cos(a.val),-a.dval*sin(a.val));}
template<class R> Diff<R> exp(const Diff<R> & a)
   { return Diff<R>(exp(a.val),a.dval*exp(a.val));}
template<class R> Diff<R> fabs(const Diff<R> & a)
   { return (a.val > 0)? a : -a;}
template<class R> bool operator<(const Diff<R> & a ,const Diff<R> & b)
   { return a.val < b.val;}
template<class R> bool operator<(const Diff<R> & a ,const R & b)
   { return a.val < b;}
```

Si l'on veut calculer l'a racine d'une equation $f(x) = y$

```
template<typename R> R f(R x)
{
   return x*x;
}
template<typename R> R Newtow(R y,R x)
{
                                                    //    solve: f(x) -y = 0
                                                //    x -= (f(x)-y) / df(x);
   while (1) {
   Diff<R> fff=f(Diff<R>(x,1));
   R ff=fff.val;
   R dff=fff.dval;
   cout << ff << endl;
   x = x - (ff-y)/dff;
   if (fabs(ff-y) < 1e-10) break;
   }
   return x;
}
```

Maintenant, il est aussi possible de re-différencier automatiquement l'algorithme de Newtow. Pour cela ; il suffit d'ecrit

```
int main()
{
    typedef double R;
    cout << " -- newtow (2)  = " << Newtow(2.0,1.) << endl;
    Diff<R> y(2.,1) ,  x0(1.,0.);                                    //    donne
    Diff<R> xe(sqrt(2.), 0.5/sqrt(2.));                          //    solution exact
    cout << "\n -- x = Newton " << y << " ," << x0 << ")"<<  endl;
    Diff<R> x= Newtow(y,x0) ;
    cout << " x = " << x << " ==  " <<  xe << " = xe    "  << endl;
    return 0;
}
```

Les resultats sont

```
[guest-rocq-135177:~/work/coursCEA/autodiff] hecht% ./a.out
 -- newtow (2)  = 1
2.25
2.00694
2.00001
2
1.41421
 -- x = Newton 2 ( d = 1)  ,1 ( d = 0) )
1 ( d = 0)
2.25 ( d = 1.5)
2.00694 ( d = 1.02315)
2.00001 ( d = 1.00004)
2 ( d = 1)
 x = 1.41421 ( d = 0.353553)  ==  1.41421 ( d = 0.353553)  = xe
```

Références

[S. Bourne] S. Bourne le système unix, InterEdition, Paris.

[Kernighhan,Pike] B.W. Kernighan, R. Pike L'environnement de programmation UNIX, InterEdition, Paris.

[1] [Kernighhan, Richie]B.W. Kernighhan et D.M. Richie Le Langage C, Masson, Paris.

[J. Barton, Nackman-1994] J. Barton, L. Nackman *Scientific and Engineering, C++*, Addison-Wesley, 1994.

[Ciaxrlet-1978] P.G. Ciarlet , *The Finite Element Method*, North Holland. n and meshing. Applications to Finite Elements, Hermès, Paris, 1978.

[Ciarlet-1982] P. G. Ciarlet *Introduction à l'analyse numérique matricielle et à l'optimisation*, Masson ,Paris,1982.

[Ciarlet-1991] P.G. Ciarlet , Basic Error Estimates for Elliptic Problems, in Handbook of Numerical Analysis, vol II, Finite Element methods (Part 1), P.G. Ciarlet and J.L. Lions Eds, North Holland, 17-352, 1991.

[2] I. Danaila, F. hecht, O. Pironneau : *Simulation numérique en C++* Dunod, 2003.

[Dupin-1999] S. Dupin *Le langage C++*, Campus Press 1999.

[Frey, George-1999] P. J. Frey, P-L George *Maillages*, Hermes, Paris, 1999.

[George,Borouchaki-1997] P.L. George et H. Borouchaki , *Triangulation de Delaunay et maillage. Applications aux éléments finis*, Hermès, Paris, 1997. Also as
P.L. George and H. Borouchaki , *Delaunay triangulation and meshing. Applications to Finite Elements*, Hermès, Paris, 1998.

[FreeFem++] F. Hecht, O. Pironneau, K. Othsuka FreeFem++ : Manual http://www.freefem.org/

[Hirsh-1988] C. Hirsch *Numerical computation of internal and external flows*, John Wiley & Sons, 1988.

[Koenig-1995] A. Koenig (ed.) : *Draft Proposed International Standard for Information Systems - Programming Language C++*, ATT report X3J16/95-087 (ark@research.att.com)., 1995

[Knuth-1975] D.E. Knuth , The Art of Computer Programming, 2nd ed., *Addison-Wesley*, Reading, Mass, 1975.

[Knuth-1998a] D.E. Knuth The Art of Computer Programming, Vol I : Fundamental algorithms, *Addison-Wesley*, Reading, Mass, 1998.

[Knuth-1998b] D.E. Knuth The Art of Computer Programming, Vol III : Sorting and Searching, *Addison-Wesley*, Reading, Mass, 1998.

[Lachand-Robert] T. Lachand-Robert, A. Perronnet (http://www.ann.jussieu.fr/courscpp/)

[Löhner-2001] R. Löhner Applied CFD Techniques, Wiley, Chichester, England, 2001.

[Lucquin et Pironneau-1996] B. Lucquin, O. Pironneau *Introduction au calcul scientifique*, Masson 1996.

[Numerical Recipes-1992] W. H. Press, W. T. Vetterling, S. A. Teukolsky, B. P. Flannery : *Numerical Recipes : The Art of Scientific Computing*, Cambridge University Press, 1992.

[Raviart,Thomas-1983] P.A. Raviart et J.M. Thomas, *Introduction à l'analyse numérique des équations aux dérivées partielles*, Masson, Paris, 1983.

[Richtmyer et Morton-1667] R. D. Richtmyer, K. W. Morton : *Difference methods for initial-value problems*, John Wiley & Sons, 1967.

[Shapiro-1991] J. Shapiro *A C++ Toolkit*, Prentice Hall, 1991.

[Stroustrup-1997] B. Stroustrup *The C++ Programming Language*, Third Edition, Addison Wesley, 1997.

[Wirth-1975] N Wirth *Algorithms + Dat Structure = program*, Prentice-Hall, 1975.

[Aho et al -1975] A. V. Aho, R. Sethi, J. D. Ullman , Compilers, Principles, Techniques and Tools, Addison-Wesley, Hardcover, 1986.

[Lex Yacc-1992] J. R. Levine, T. Mason, D. Brown Lex & Yacc, O'Reilly & Associates, 1992.

[Campione et Walrath-1996] M. Campione and K. Walrath The Java Tutorial : Object-Oriented Programming for the Internet, Addison-Wesley, 1996.
Voir aussi Integrating Native Code and Java Programs. `http://java.sun.com/nav/read/Tutorial/native1.1/index.html`.

[Daconta-1996] C. Daconta Java for C/C++ Programmers, Wiley Computer Publishing, 1996.

[Casteyde-2003] Christian Casteyde Cours de C/C++ `http://casteyde.christian.free.fr/cpp/cours`

[3] C. Berge, *Théorie des graphes*, Dunod, 1970.